UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Information Technology Curriculum

Taavi Kala

Efficient discovery of Declare models from event logs

Bachelor's Thesis (6 ECTS)

Supervisor: Fabrizio Maria Maggi

Tartu 2015

Efficient discovery of Declare models from event logs

Abstract:

The purpose of the following thesis is to improve the performance of a process mining tool called DeclareMiner. The DeclareMiner represents process models in a declarative modelling language called Declare which is widely used in process mining due to its readability and understandability. The current version of the Declare Miner was underperforming in some cases. To improve the performance of the tool, an approach from [1] has been integrated into the original algorithm. The result is a new, standalone application whose performance is multiple times better than the existing one. In addition, the new implementation is able to provide different outputs among which a user-friendly and readable report, understandable to people not expert of process mining and Declare.

Keywords:

Process mining, Declare modelling language, event logs, ProM, DeclareMiner, replayers

Tõhus Declare mudelite avastamine sündmuste logidest

Lühikokkuvõte:

Käesoleva lõputöö eesmärgiks on parandada protsesside kaevandamise tööriista DeclareMiner jõudlust. DeclareMiner väljendab protsesside mudeleid deklaratiivses modelleerimiskeeles nimega Declare, mis on protsessikaevadamise maailmas laialt levinud oma loetavuse ja arusaadavuse tõttu. Olemasoleva DeclareMiner versiooni jõudlus oli mõningatel juhtudel ebapiisav. Jõudluse parandamiseks kasutati artiklis [1] arendatud lähenemist, mis integreeriti tööriista originaal algoritmi. Tulemuseks on uus, eraldiseisev rakendus, mille jõudlus on olemasolevast lahendusest mitu korda parem. Samuti suudab uus rakendus koostada erinevaid väljundeid, millest üks on kasutajasõbralik ning loetav ka neile, kes pole protsesside kaevandamise ja Declare keele eksperdid.

Võtmesõnad:

Protsesside kaevandamine, Declare modelleerimiskeel, sündmuste logid, ProM, DeclareMiner, replayerid

Table of Contents

1 Introduction

Process mining is a family of techniques that allow for the analysis of business processes. Its main focus lies in the automatic retrieval and subsequent analysis of business process models from event logs. Process mining consists of discovery, enhancement and conformance checking [2]. Discovery is the extraction of process models from an event log. Enhancement is the extension or improvement of process models using information extracted from a log. Conformance checking consists of analyzing whether the reality, as recorded in a log, is compliant with a process model [3].

The majority of process discovery algorithms try to construct a procedural model. However, the resulting models are often spaghetti-like and difficult to interpret especially for processes working in unstable environments. Therefore, it is useful to discover declarative process models instead when dealing with processes with a lot of variability and where multiple paths are allowed. An approach is presented in [4] to discover Declare models from event logs. Declare is a declarative process modelling language first introduced in [5]. In [4], a plugin is presented called DeclareMiner, developed for the process mining tool ProM using automata and an apriori algorithm for model detection.

The approach proposed in this thesis integrates the approach presented in [1] into the existing implementation developed in [4]. In addition the usability of the mining results has been improved as well.

As a result of this thesis the DeclareMiner runs faster than the original version and provides richer results with respect to it. The code developed as a contribution of this thesis is provided as an open source application.

2 State of the art

In this chapter related work will be discussed to give a better overview of the general idea of this thesis.

Probabilistic declarative process is an approach based on Statistical Relational Learning for analyzing a log containing several traces of a process labeled as compliant or non-compliant. Based on that it is possible to learn a set of declarative constraints expressed as ICs (Integrity Constraints) which are represented in Markov Logic [6]. A logic-based approach for probabilistic process mining enhancing this approach is presented in [7].

An approach that makes use of logical programming for declarative process mining is presented in [8]. The proposed methodology is based on Inductive Logical Programming (ICL). The ICL algorithm, used in this approach, is adapted to the problem of learning integrity constraints in SCIFF and is able to learn a model by considering both compliant and non-compliant traces.

A component to discover declarative processes was developed for ProM in [4]. The component is called DeclareMiner. It uses an apriori algorithm to build a list of candidate constraints to be discovered. This list is pruned by checking the constraints against the log.

An algorithm to discover declarative workflows was developed in [9] using email messages as event log traces. The algorithm implemented is called MINERful++ which is described as a two-step algorithm. The first step is to build the knowledge base based on the given traces. The second step is to compute the statistical support of constraints by querying the knowledge base.

An online process discovery technique which takes data from online streams is presented in [1]. The proposed framework is able to produce at runtime an updated picture of the process behavior in terms of Declare constraints. It also gives meaningful information about the concept drifts that occurred during the process execution to the user.

3 Background

3.1 Process mining

Process mining is still a rather young research discipline which lies between data mining and computational intelligence as well as process modelling and analysis. The general idea of process mining is to discover, monitor and improve real life processes by extracting knowledge from actual event logs used in different systems that gather event data. Over the last ten years, event data have become more widely available and process mining techniques have matured a lot. Different process mining algorithms have been implemented in academic and commercial systems as there is in increasing interest from industry in process mining. Thus, an increasing amount of software vendors are adding functionalities that provide process mining capabilities to their software and tools.

The main branches of process mining are

- *process discovery* which takes an event log and produces a model without using any apriori information
- *conformance checking* which is used to compare the existing process model with an event log of the same process
- *model extension* which is used to extend existing models with information coming from logs
- *model repair* which is used to repair already existing models using event logs

A list of guiding principles that aid in avoiding mistakes that can be made when applying process mining in actual, real-life settings is presented in [3]. The guide consists of the following six principles:

- *Event data should be treated as first-class citizens,* which means that the event logs are classified under different maturity levels ranging from excellent to poor or 5 stars to 1 start respectively. The higher the maturity level, the more reliable are the results when process mining is applied to the log
- *Log extraction should be driven by questions* because without concrete questions it is very difficult to extract reasonable information from event logs
- *Concurrency, choice and other basic control-flow constructs should be supported* to make sure the generated models are fitting and easy to understand

- *Events should be related to model elements* in order to support conformance checking and enhancement
- *Models should be treated as purposeful abstractions of reality* due to the fact that the results may be used by various stakeholders in different situations. It also helps with producing understandable maps
- *Process mining should be a continuous process* to cope with process changes

Along-side key points and guide principles, there are challenges that need to be addressed due to the fact that process mining is, as mentioned before, a young discipline. These challenges are considered to be incomplete as, over time, new challenges may appear or existing challenges may disappear due to advances is process mining. Nevertheless, the challenges listed below are still relevant [3].

- *Finding, merging, and cleaning event data*
- *Dealing with complex event logs having diverse characteristics*
- *Creating representative benchmarks*
- *Dealing with concept drift*
- *Improving the representational bias used for process discovery*
- *Balancing between quality criteria such as fitness, simplicity, precision and generalization*
- *Cross-organizational mining*
- *Providing operational support*
- *Combining process mining with other types of analysis*
- *Improving usability for non-experts*

An event log is the key element for any process mining technique – in an event log:

- *Each event refers to an activity* (a well-defined step in the process)
- *Each event refers to a trace* (a process instance)
- *Each event can have a performer also referred to as originator* (the actor executing or initiating the activity)
- *Events have a timestamp and are totally ordered*

Since each information system has its own format for storing log files a generic XML format to store in a log information about process executions called MXML has been developed. The MXML format is presented in figure 1 (a). A log file typically contains information about events that took place in a system (*AuditTrailEntry* in XML). Such events typically refer to a trace *(ProcessInstance* in the XML) and a specific activity (*WorkflowModelElement* in XML) within that trace. The originator and the timestamp are connected to the *AuditTrailEntry* so they are always related to the event itself. Figure 1 (b) shows the transactional model for activity lifecycles. The transactional model is adopted by several commercial systems [10].

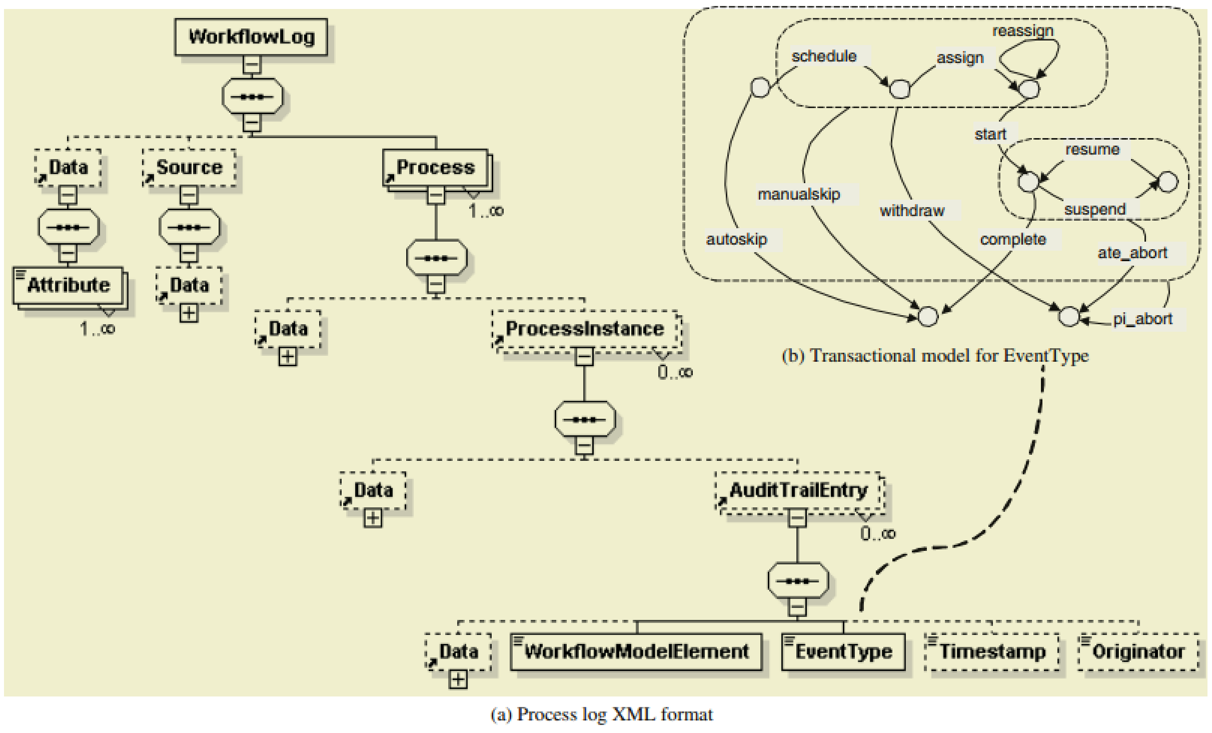


Figure 1. Process log XML format (a) and transactional model (b)

Even if MXML has been used as a standard for storing event logs for several years, based on practical experiences with applying MXML in about one hundred organizations, several problems and limitations related to MXML format have been discovered. One of the main problems is the semantics of additional attributes stored in the event log. In MXML, these are all treated as string values with a key and have no generally understood meaning. Another problem is the nomenclature used for different concepts. This is caused by the MXML's assumption that strictly structured process would be stored in this format.

To solve the problems encountered with MXML, and to create a standard that could also be used to store event logs from many different information systems directly, a new format has been developed called eXtensible Event Stream or XES. It enhances the MXML format in many ways as shown in [11]. The XES *Log* element replaces the MXML *WorfklowLog* element, the *Trace* element replaces the *ProcessInstance* element, and the *Event* element replaces the *AuditTrailEntry* element. However, there are a number of differences worth mentioning. First of all, in XES the *Log*, *Trace* and *Event* elements only define the structure of the document: they do not contain any information themselves. To store any data in the XES format, attributes are used. Every attribute has a string based key, a known type, and a value of that type. Possible types are string, date, integer, float and boolean. Note that attributes can have attributes themselves which can be used to provide more specific information [11]. The meta-model of XES is shown in figure 2. The advantages of XES are simplicity, flexibility, extensibility and expressivity. From these points of view it improves MXML.

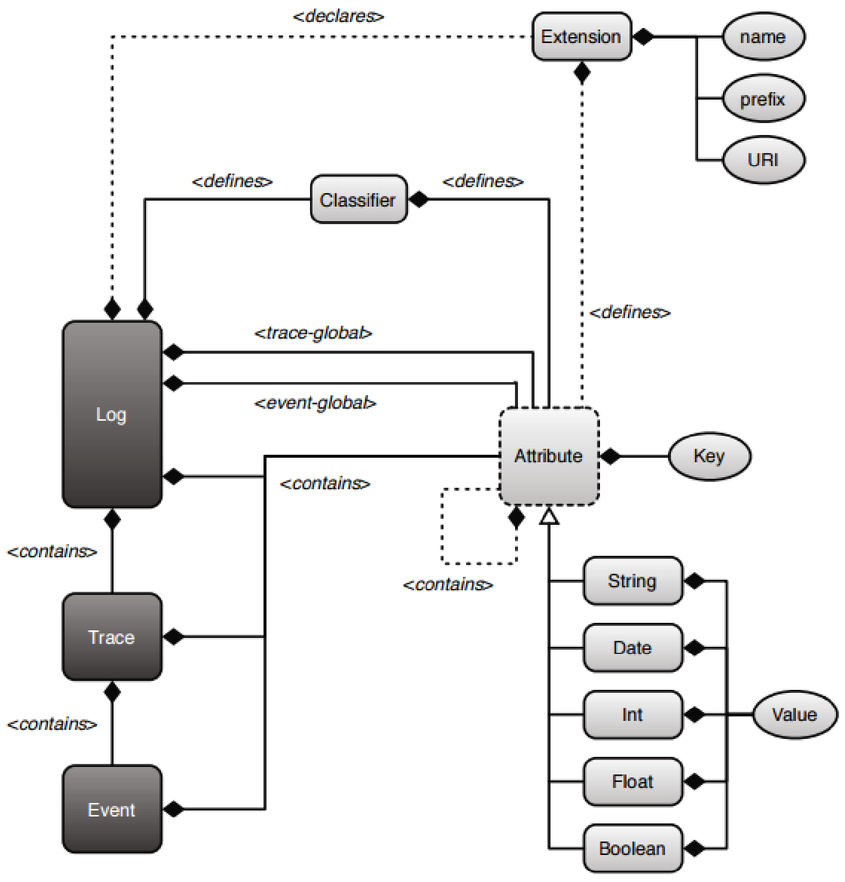


Figure 2: XES meta-model

3.2 Declare modeling language

Declare is a declarative process modelling language and a constraint based system that uses Linear Temporal Logic (LTL) for the development and execution of process models [5]. The three main components it consists of are

1. *Designer* (modeling tool), that is used for system settings and process model design
2. *Framework* (process enactment tool), which is also used for communication with other programs and changing models at run-time
3. *Worklist* (process execution tool), which is meant for users to execute process instances and see recommendations

Different application domains may require a different set of relation types (constraint templates). Therefore, Declare facilitates the definition of sets of constraint templates.

Compared to the imperative models, a declarative model defines a set of constraints that should be followed during the execution. In that way, a declarative model implicitly defines the control-flow as all possibilities that do not violate any of the given constraints. In this way declarative models, differently from the imperative ones, enjoy flexibility. Each constraint template has three attributes:

1. *A unique name*
2. *Semantics specified in LTL*
3. *Graphical representation* (for visual representation)

There is a total of 19 different templates in the Declare modeling language. 5 of these are existence templates which involve only one event. 11 are relation templates which describe a dependency between two events and 3 are negative relation templates [12].

The existence templates are the following:

- *existence(n, A)* which specifies that *A* should occur *at least n* times in a process instance
- *absence(n +1, A)* which specifies that *A* should occur *at most n* times
- *exactly(n, A)* which specifies that *A* should occur *exactly n* times
- *init(A)* which specifies that each process instance should start with event *A*

The LTL semantics and graphical representations of the existence templates can be seen from table 2. The LTL operator semantics can be seen from table 1.

The relational templates are the following:

- *responded existence(A, B)* which specifies that if event *A* occurs, event *B* should also occur
- *co-existence(A, B)* which specifies that if one of the events *A* or *B* occurs, the other one should also occur
- *response(A, B)* which specifies if event *A* occurs, event *B* should eventually occur after *A*
- *precedence(A, B)* which specifies that event *B* should occur only if event *A* has occurred before
- *succession(A, B)* which requires both *precedence* and *response* algorithms to hold between events *A* and *B*
- *alternate response(A, B)* which is the same as *response* but allows no repetitions of these events in between
- *alternate precedence(A, B)* which is the same as *precedence* but allows no repetitions of these events in between
- *alternate succession(A, B)* which is the same as *succession* but allows no repetitions of these events in between
- *chain response(A, B)* which is the same as *response* but the events must happen one after another
- *chain precedence(A, B)* which is the same as *precedence* but the events must happen one after another
- *chain succession(A, B)* which is the same as *succession* but the events must happen one after another

The LTL semantics and graphical representations of the relational templates can be seen from table 3.

The negative relation templates are the following:

- *not co-existence(A, B)* which specifies that *A* and *B* cannot occur together in the same process
- *not succession(A, B)* which specifies that that any occurrence of *A* cannot be eventually followed by *B*
- *not chain succession(A, B)* which specifies that *A* cannot be directly followed by *B*

The LTL semantics and graphical representations of the negative relation templates can be seen from table 4.

operator	***semantics***
$\bigcirc\varphi$	*φ has to hold in the next position of a path.*
$\square\varphi$	*φ has to hold always in the subsequent positions of a path.*
$\lozenge\varphi$	*φ has to hold eventually (somewhere) in the subsequent positions of a path.*
$\varphi\, U \psi$	*φ has to hold in a path at least until ψ holds. ψ must hold in the current or in a future position.*

Table 1: LTL operators' semantics

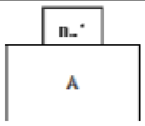
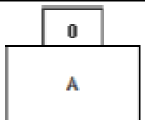
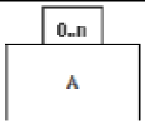
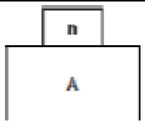
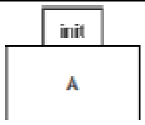
name of template	***LTL semantics***	***graphical representation***
$existence(1, A)$ $existence(2, A)$ ... $existence(n, A)$	$\lozenge A$ $\lozenge(A \wedge \bigcirc(existence(1, A)))$ ... $\lozenge(A \wedge \bigcirc(existence(n-1, A)))$	
$absence(A)$	$\neg existence(1, A)$	
$absence(2, A)$ $absence(3, A)$ ... $absence(n+1, A)$	$\neg existence(2, A)$ $\neg existence(3, A)$ ... $\neg existence(n+1, A)$	
$exactly(1, A)$ $exactly(2, A)$ ... $exactly(n, A)$	$existence(1, A) \wedge absence(2, A)$ $existence(2, A) \wedge absence(3, A)$ ... $existence(n, A) \wedge absence(n+1, A)$	
$init(A)$	A	

Table 2: Existence templates

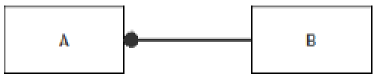
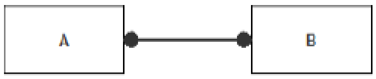
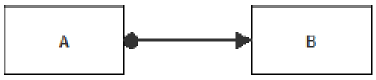
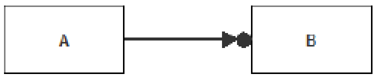
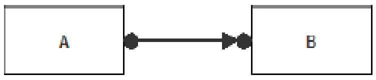
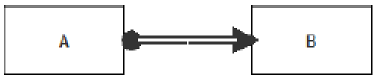
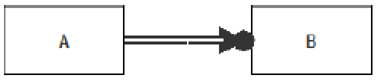
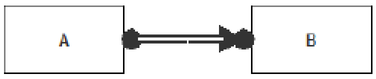
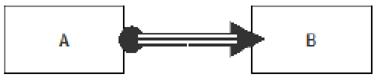
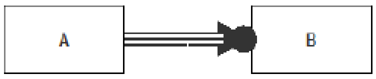
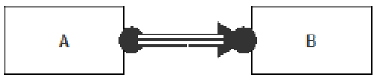
name of template	*LTL semantics*	*graphical representation*
$responded\ existence(A, B)$	$\Diamond A \Rightarrow \Diamond B$	
$co\text{-}existence(A, B)$	$\Diamond A \Leftrightarrow \Diamond B$	
$response(A, B)$	$\Box(A \Rightarrow \Diamond B)$	
$precedence(A, B)$	$(\neg B\ U A) \vee \Box(\neg B)$	
$succession(A, B)$	$response(A, B) \wedge$ $precedence(A, B)$	
$alternate\ response(A, B)$	$\Box(A \Rightarrow \bigcirc(\neg A\ U B))$	
$alternate\ precedence(A, B)$	$precedence(A, B) \wedge$ $\Box(B \Rightarrow \bigcirc(precedence(A, B)))$	
$alternate\ succession(A, B)$	$alternate\ response(A, B) \wedge$ $alternate\ precedence(A, B)$	
$chain\ response(A, B)$	$\Box(A \Rightarrow \bigcirc B)$	
$chain\ precedence(A, B)$	$\Box(\bigcirc B \Rightarrow A)$	
$chain\ succession(A, B)$	$\Box(A \Leftrightarrow \bigcirc B)$	

Table 3: Relation templates

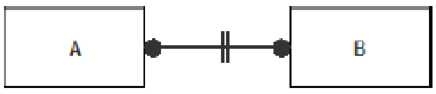
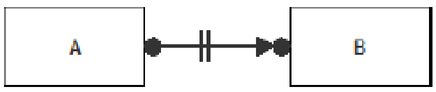
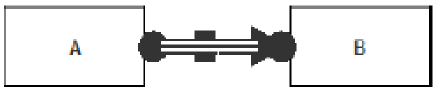
name of template	*LTL semantics*	*graphical representation*
not co-existence(A, B)	$\neg(\Diamond A \wedge \Diamond B)$	
not succession(A, B)	$\Box(A \Rightarrow \neg(\Diamond B))$	
not chain succession(A, B)	$\Box(A \Rightarrow \bigcirc(\neg B))$	

Table 4: Negative relation templates

A Declare model containing multiple constraints is defined as a conjunction of the constraints. This means that the actions of users during execution must fulfill all constraints. Declare constraints can be either mandatory or optional.

The system forces its users to follow all mandatory constraints in the model. In case of optional constraints users have the ability to decide whether to follow the corresponding rule or to violate it. Optional constraints are not enforced by the Declare system during execution. When a user is about to perform an action that violates an optional constraint, a warning about the violation is presented and the user can decide whether to continue with the action and violate the constraint or to cancel the action and follow the constraint. The text of the warning can be specified in the definition of the constraint.

A model in Declare is mapped onto a set of LTL formulas. Based on these LTL formulas, automata are automatically generated to support enactment. Declare uses an algorithm that creates finite-words automata from LTL formulas of the constraints that are used. These automata are used both to drive the execution and to monitor the state of each constraint.

Some compositions of constraints in process models may cause errors that lead to problems at run-time. Thus, Declare verifies process models against different types of errors and finds a minimal set of constraints that causes a specific error. All models can be verified against dead activities and conflicting constraints. A dead activity is an activity that can never be executed in the model. A set of constraints is conflicting if there exists no execution that would fulfill all constraints [13].

3.3 DeclareMiner plugin in the ProM framework

Prom framework

ProM is a process mining framework that integrates the functionality of several existing process mining tools and provides many additional process mining plug-ins. It supports multiple formats and multiple languages such as Petri nets, EPCs, Social Networks and so on and so forth. The plug-ins can be used in several ways and combined to be applied in real-life situations [10].

DeclareMiner plugin

The Declare Miner plug-in for ProM allows users to discover a Declare model from a log by specifying a number of settings. There are two versions of the plug-in. The first one, the Declare Miner, requires a user-specified Declare language as input. The second one, the Declare Miner Default, uses a predefined Declare language and does not require any language as input. More information on the usage and set-up can be found at http://www.win.tue.nl/declare/declare-miner/ .

4 Proposed Contribution

The algorithm proposed in this thesis combines the apriori algorithm from [4] and the replayers from [1] to detect the models of interest. The algorithm is composed of two phases. In phase one a list on candidate constraints is generated using an apriori algorithm. In the second phase the list of candidate constraints is pruned using the approach presented in [1]. The proposed algorithm provides very good efficiency.

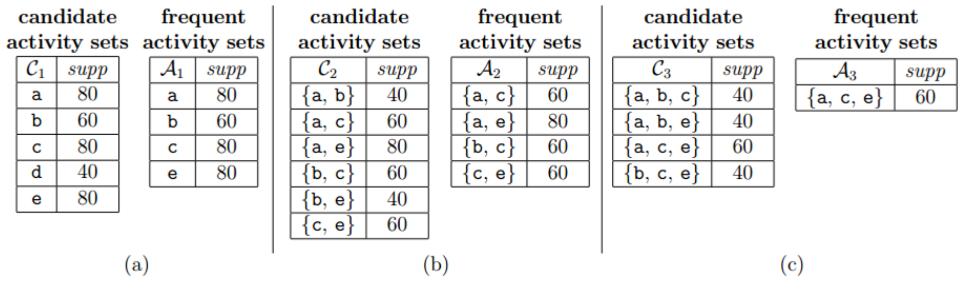
4.1 Apriori algorithm

Firstly, the frequency of single activities is computed as the support value (frequency of activity sets of length 1) for a given activity. After that, the activities with a support value higher than the minimum support are considered. Then, the considered activities are combined into pairs where both of the activities have support higher than the minimum support. Next, the frequency corresponding to the percentage of traces in which both activities of a pair occur in the same trace are computed as the support value (frequency of activity sets of length 2) of the given pair of activities. After that, only the pairs with support value higher than the minimum support are considered. From those considered activities, a list of candidate constraints is instantiated by instantiating standard Declare templates using the identified pairs.

An example sets of candidates and the frequent activity sets for log

$$\mathcal{L} = [(e,a,b,a,a,c,e,),(e,a,a,b,c,e),(e,a,a,d,d,e),(\ b,b,c,c),(\ e,a,a,c,d,e)]$$

based on minimum support 50% can be seen from figure 3.



(a)

candidate activity sets

$\mathcal{C}_1$	*supp*
a	80
b	60
c	80
d	40
e	80

frequent activity sets

$\mathcal{A}_1$	*supp*
a	80
b	60
c	80
e	80

(b)

candidate activity sets

$\mathcal{C}_2$	*supp*
{a, b}	40
{a, c}	60
{a, e}	80
{b, c}	60
{b, e}	40
{c, e}	60

frequent activity sets

$\mathcal{A}_2$	*supp*
{a, c}	60
{a, e}	80
{b, c}	60
{c, e}	60

(c)

candidate activity sets

$\mathcal{C}_3$	*supp*
{a, b, c}	40
{a, b, e}	40
{a, c, e}	60
{b, c, e}	40

frequent activity sets

$\mathcal{A}_3$	*supp*
{a, c, e}	60

Figure 3: Discovering frequent activity sets using the apriori algorithm in the event log $\mathcal{L}$. The support values are expressed in %.

4.2 Replayers algorithm

After the list of candidate Declare constraints has been detected in phase one, the list is pruned using template replayers. Each template replayer implements a different algorithm to compute the support of a candidate constraint corresponding to the given template. The support of a constraint is the percentage of traces in which the constraint is satisfied. For each desired template to be discovered a replayer instance is initialised with the list of candidate constraints corresponding to that template. Then, the log is iterated and each event in the log is processed by each replayer. After all the replayers have finished processing the events in the log, the candidate constraints with support lower than the minimum support are discarded. The remaining ones are added to the final Declare model to be presented to the user. The Response, Precedence and Existence constraint replayer algorithms are briefly explained in the following sections.

Response replayer

The semantics of a $Response$ constraint can be defined as follows - whenever activity a is executed, activity b is eventually executed afterwards. The pseudo code for the Response replayer can be seen from Algorithm 1. A brief explanation of the pseudo code is below:

1. Define maps $fulfilledCases$ and $pendingActivations$ unless they are already defined
2. For each activity pair k_1 and k_2 in $candidateList$, the incoming activity a is compared to k_2
   - 2.1. If k_2 is the same activity as a, the *Response* constraint (k_1, k_2) has no pending activations because a happened after k_1
   - 2.2. If k_1 is the same activity as a, the number of pending activations of the *Response* constraint (k_1,k_2) is incremented by 1 unit because a is waiting for an occurrence of k_2
   - 2.3. In case of last event, get the pending activations for activity pair k_1 and k_2.
     - 2.3.1. If the number of pending activations is 0, the number of fulfilled cases for activity pair k_1 and k_2 is increased because no pending activations were detected

```
Input: candidateList the candidate list from phase one
       e =(c,a,t) the event to be processed (c is the case id of the event,
       a is the activity name, t is the timestamp)
       isLastEvent determines if the current events is the last event of
       the case

if fulfilledCases is not defined then define map fullfilledCases
if pendingActivations is not defined then define map pendingActivations
foreach (k1,k2) in candidateList do
    if k2 = a then
      pendingActivations.put((k1,a),0)
    else if k1 = a then
      acts <- pendingActivations.get((a,k2))
      pendingActivations.put((a,k2), acts+1)
    if isLastEvent do
      acts <- pendingActivation.get((k1,k2))
      if acts = 0 do
        cases <- fulfilledCases.get((k1,k2))
        fulfilledCases.put((k1,k2), cases+1)
```

Algorithm 1: Response constraint pseudo code

Precedence replayer

The semantics of a $Precedence$ constraint can be defined as follows – Activity a is preceded by activity b. Activity b happens only after activity a had happened. The pseudo code for the Precedence replayer can be seen from Algorithm 2. A brief explanation of the pseudo code is the following:

1. Define map $fullfilledCases$ and set $activityFrequencies$ and $fulfilledActivations$ unless they are already defined
2. If $activityFrequencies$ contains activity a, increment its frequency
3. Iterate over $candidateList$ list pairs (k_1, k_2). Compare a to k_1
    3.1. If k_2 is the same activity as a and k_1 frequency in set $activityFrequencies$ is greater than 0, the number of fulfilled activations of the $Precedence$ constraint (k_1, k_2) is incremented by 1 unit in $fulfilledActivations$ because a happened before k_2
    3.2. In case of last event, get the number of fulfilled activations for k_1 and k_2 from $fulfilledActivations\ map$
        3.2.1. If the number of fulfilled activations is the same as the number of occurrences of k_2 the number of fulfilled cases for activity pair k_1 and k_2 is incremented because all the activations are fulfilments

```
Input: candidateList the candidate list from phase one
       e =(c,a,t) the event to be processed (c is the case id of the event,
       a is the activity name, t is the timestamp)

if fulfilledCases is not defined then define map fullfilledCases
if activityFrequencies is not defined then define map activityFrequencies
if fulfilledActivatons is not defined then define map fulfilledActivations
if activityFrequencies.constaintsKey(a)
  freq <- activityFrequencies.get(a)
  activityFreqencies.put(a, freq+1)
foreach (k1,k2) in candidateList do
  if k2 = a && activityFrequencies.get(k1) > 0 then
      acts <- fulfilledActivations.get((a,k2))
      fulfilledActivations.put((a,k2), acts+1)
  if isLastEvent do
    acts <- fulfilledActivations.get((k1,k2))
    if acts = activityFrequecies.get(k2) do
      cases <- fulfilledCases.get((k1,k2))
      fulfilledCases.put((k1,k2), cases+1)
```

Algorithm 2: Precedence constraint pseudo code

Existence replayer

The $Existence(n)$ constraint can be described as such – Activity a is executed at least n times. The $Absence(n)$ constraint description is as follows – Activity a is executed at most $n-1$ times. The $Exactly(n)$ constraint can be described as follows – Activity a is executed exactly n times. The pseudo code for the Existence, Exactly and Absence replayers can be seen from Algorithm 3. A brief explanation of the pseudo code can be explained as such:

1. Define map $fulfilledCases$ and set $activityfrequencies$
2. If $activityFrequencies$ contains activity a, increment its frequency
3. Iterate over $candidateList$ list items K
   3.1. In case of last event, get the number of activity frequencies for k from
      3.1.1. If the existence condition for activity frequencies for k holds, the number of fulfilled cases for k is incremented

The list of different existence conditions are:

- *Existence(n)* - the frequency must be greater than or equal to *n*
  - Possible constraints are Existence, Existence2 and Existence3
- *Absence(n)* - the frequency must be at most *n-1*
  - Possible constraints are Absence, Absence2 and Absence3
- *Exactly(n)* - the frequency must be exactly *n*
  - Possible constraints are Exactly1 and Exactly2

```
Input: candidateList the candidate list from phase one
       e =(c,a,t) the event to be processed (c is the case id of the event,
       a is the activity name, t is the timestamp)


if fulfilledCases is not defined then define map fullfilledCases
if activityFrequencies is not defined then define map activityFrequencies
if activityFrequencies.constaintsKey(a)
  freq <- activityFrequencies.get(a)
  activityFreqencies.put(a, freq+1)
foreach (k) in candidateList do
  if isLastEvent do
    acts <- activityFrequencies.get(k)
    if existenceCondition(acts) do
      cases <- fulfilledCases.get(k)
      fulfilledCases.put(k, cases+1)
```

Algorithm 3: Existence (and similar constraints) pseudo code

4.3 The final result

As mentioned in the introduction, the result of this thesis is an application that improves of the existing DeclareMiner plugin for ProM. The application that has been developed for this thesis is intended to be run for a command line interface such as Unix Terminal or Windows Command Line. The application is available open source as a JAR file which makes it easier to integrate into Java web applications for example. The application retains the initial functionality of the DeclareMiner plugin but with improved performance and richer outputs.

The only input it gets is a configuration file as an absolute file path where one can specify the following arguments set in the format of *variable=value*. The arguments to be used are:

1. *log_file_path* which is the path of the log file to be processed (using either XES or MXML as an extension, compressed log files with an extra GZ extension are also accepted)
2. *templates* as a comma separated string of desired template names (e.g. Succession, Response, Precedence etc.)
3. *min_support* which is the minimum support used for constraint filtering (ranges between 0 to 100)

4. *alpha* which is the alpha value also used for constraint filtering (either 0 or 100). Shortly put, it enables (0) or disables (100) the vacuity detection. If the vacuity detection is enabled only constraints that are activated and satisfied frequently will be discovered. If vacuity detection is disabled, also vacuously satisfied constraints will be discovered. For example, for a $Response$ constraint, requiring that every occurrence of a must be followed by an occurrence of b, all the traces that do not contain any occurrence of a trivially (or vacuously) satisfy the constraint.
5. *output_path* to specify the absolute path of the output file
6. *output_file_type* which is used to select the desired format of the output file. Possible values are XML, TEXT and REPORT which are explained below.

To run the jar file it must be called using the following command:

```
java –cp declare_miner.jar BetterMiner /path/to/configuration_file
```

The declare_miner.jar is the file to be executed. BetterMiner is the class that is used to start the mining job and the /path/to/configuration_file is an example path where the configurations file is stored.

The output it provides is one of three different type of files:

1. *XML* which is an XML file formatted to be read by the Declare Designer
2. *TEXT* that features a simple list of constraints discovered with the corresponding support
3. *REPORT* which is a human readable output that can be understood by a person not expert of process mining and Declare. It uses simple and logical sentences to convey the constraint's essence. For every constraint it also lists the witnesses (cases in which the constraint is satisfied), counter examples (cases in which the constraint is violated) and vacuous cases.

An example of the text file are the following lines:

```
Response(Receive Order-complete, Send Invoice-complete): 1.0
Response(Receive Order-complete, Receive Payment-complete): 1.0
Response(Receive Payment-complete, Send Invoice-complete): 1.0
Response(Receive Payment-complete, Ship Products-complete): 1.0
Response(Receive Order-complete, Ship Products-complete): 1.0
Response(Ship Products-complete, Send Invoice-complete): 1.0
```

An example of the report are these sentences:

```
Whenever activity 'Receive Order-complete' is executed, activity
'Send Invoice-complete' is eventually executed afterwards.

witnesses (100,00% of cases, 3 cases in total): 1, 2, 3
counter examples (0,00% of cases, 0 cases in total):
vacuous cases (0,00% of cases, 0 cases in total):

Whenever activity 'Receive Order-complete' is executed, activity
'Receive Payment-complete' is eventually executed afterwards.

witnesses (100,00% of cases, 3 cases in total): 1, 2, 3
counter examples (0,00% of cases, 0 cases in total):
vacuous cases (0,00% of cases, 0 cases in total):
```

The source code for the application can be accessed using the following hyperlink https://bitbucket.org/taavi_kala/declareminerwithaprioriandreplayers/src

5 Benchmarks

In our experiments, the old implementation of the DeclareMiner using apriori and automata was compared to approach with replayers presented in [1] and the new implementation proposed in this thesis using apriori and replayers. The time was recorded in milliseconds and it corresponds to the start and end time of pruning part of the whole algorithm. The old implementation is referred to as “automata + apriori”, the replayers implementation as “replayers” and the new version as “replayers + apriori” in the graphs.

The benchmarking tests were carried out on multiple different synthetic logs and real-life logs that were provided for the BPI Challenges in years 2012, 2013 and 2014.

The tests were run using configurations where different minimum support values (60, 70, 80, 90 and 100) and alpha values (0 and 100 i.e. enabled and disabled vacuity detection) were used. This was done for each template separately and after that for all templates together as the *templates* argument in the configuration file can handle comma separated values. The overall results show that the new implementation outperforms the other two.

The scripts for generating logs and running the tests can be found at the project source code (generate-logs.rb and tests-runner.rb).

The machine where the tests were run on is a KVM (Kernel-based Virtual Machine) and has the following specifications:

- CPU – 8 cores, 2.5 GHz each
- RAM – 32 GB

5.1 Synthetic logs

Synthetic logs were generated using the generator described in [14] that has the ability to generate logs for a combination of given command line arguments. The log contents were generated based on three different arguments:

1. *Log size* which determines how many different traces will be generated into the log. The values used were 100, 200, 500, 700, 1000, 2000, 5000 and 7000.
2. *Trace length* which determines the number of events inside a single trace. Values used were 5, 10, 20, 25 and 30. The minimum and maximum trace lengths was equally valued to keep the logs consistent so each trace had exactly *n* events.
3. *Alphabet size* which determines the number of different activities to be generated into the log. The values used were 5, 10, 15, 20 and 25 where each of the numbers mean a number of colon-separated Latin alphabetical characters inside a string (A:B:C:D:E for 5, A:B:C:D:E:F:G:H:I:J for 10 etc.)

Synthetic logs were generated in the following way - while changing one argument, other arguments were using fixed values i.e. while generating logs based on different log size the trace length was fixed to 15 and alphabet size to 20. The fixed value for log size while generating different trace lengths and alphabet sizes was 1000.

Varying the log size

For these logs the results are presented for different values of minimum support and alpha. Other values are fixed to alphabet size 20, trace length 15 and the results are taken from configuration where all templates are benchmarked together

In general it can be seen that the performance of the replayers and apriori, in case of vacuity detection enabled, is much faster compared to the automata and apriori. After disabling the vacuity detection the difference is even more evident.

Minimum support 80% and 90% results show little difference to previous minimum support values. The more noticeable observation is that the replayers only implementation becomes less efficient than automata and apriori for log sizes 5000 and 7000 (figures 6 and 7).

In case of minimum support 100% the replayers and apriori does perform better than the automata and apriori but less significantly especially in case of enabled vacuity detection.

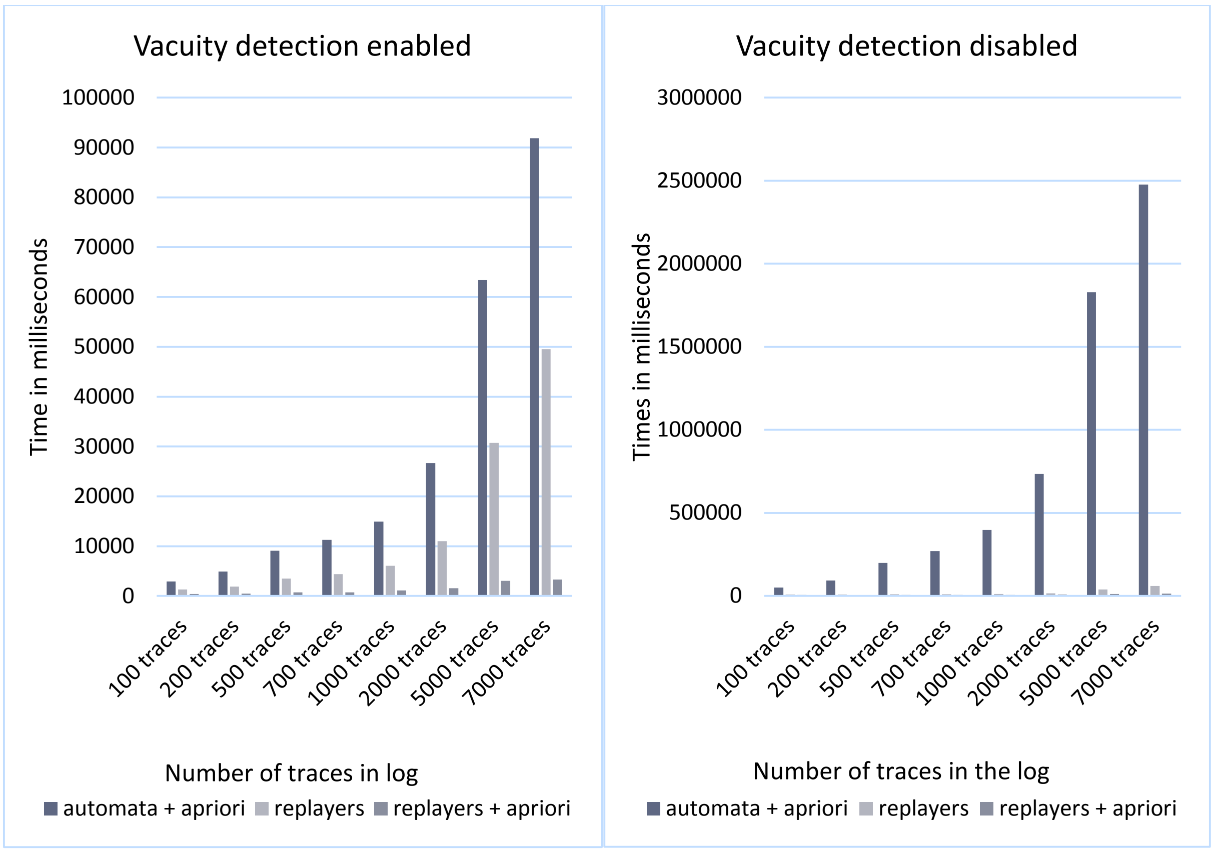


Figure 4: Log size results for minimum support 60%

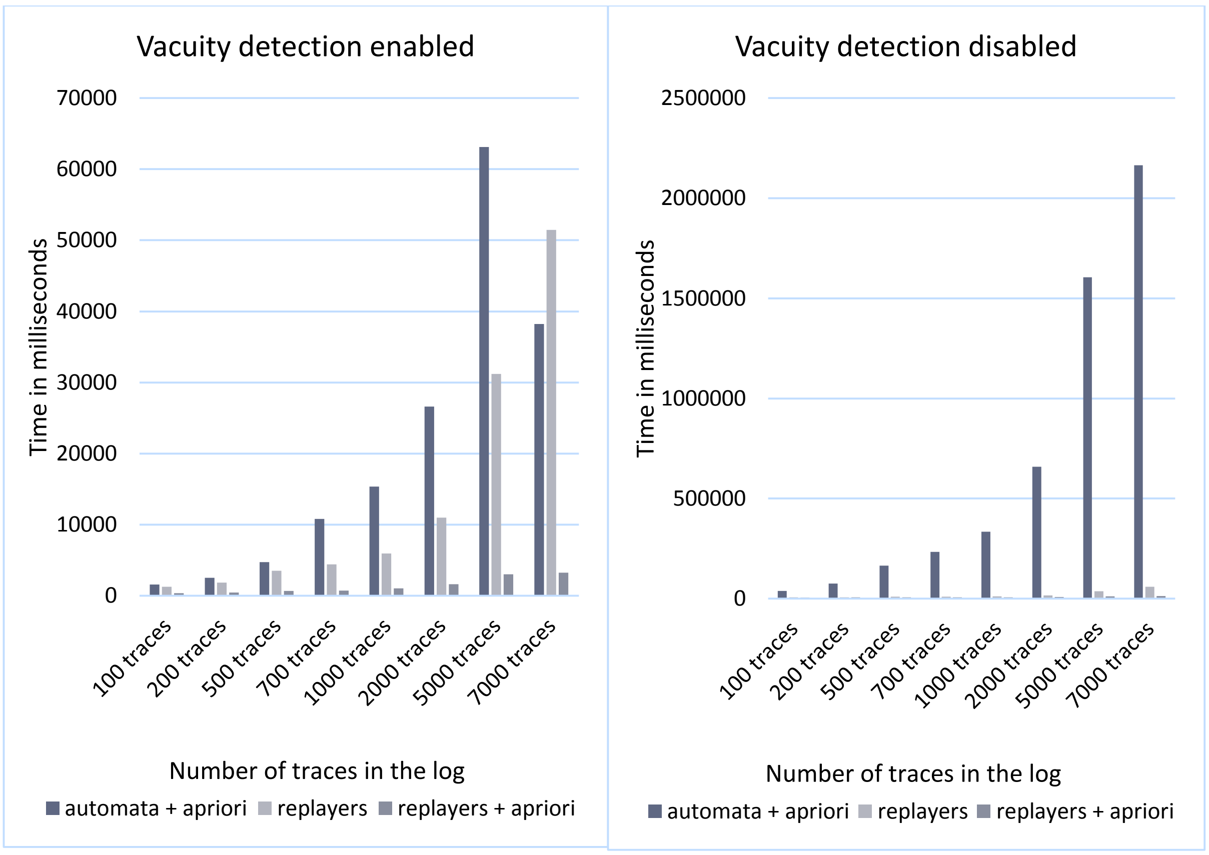


Figure 5: Log size results for minimum support 70%

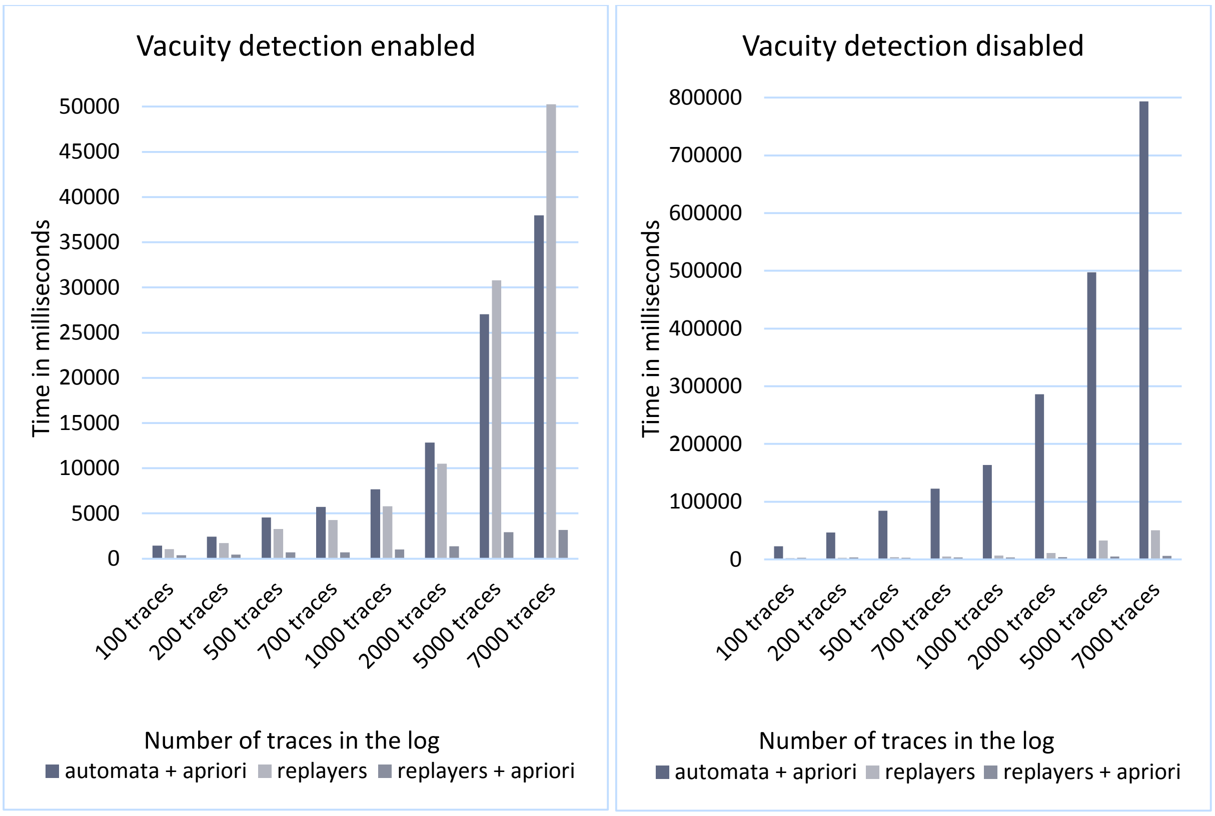


Figure 6: Log size results for minimum support 80%

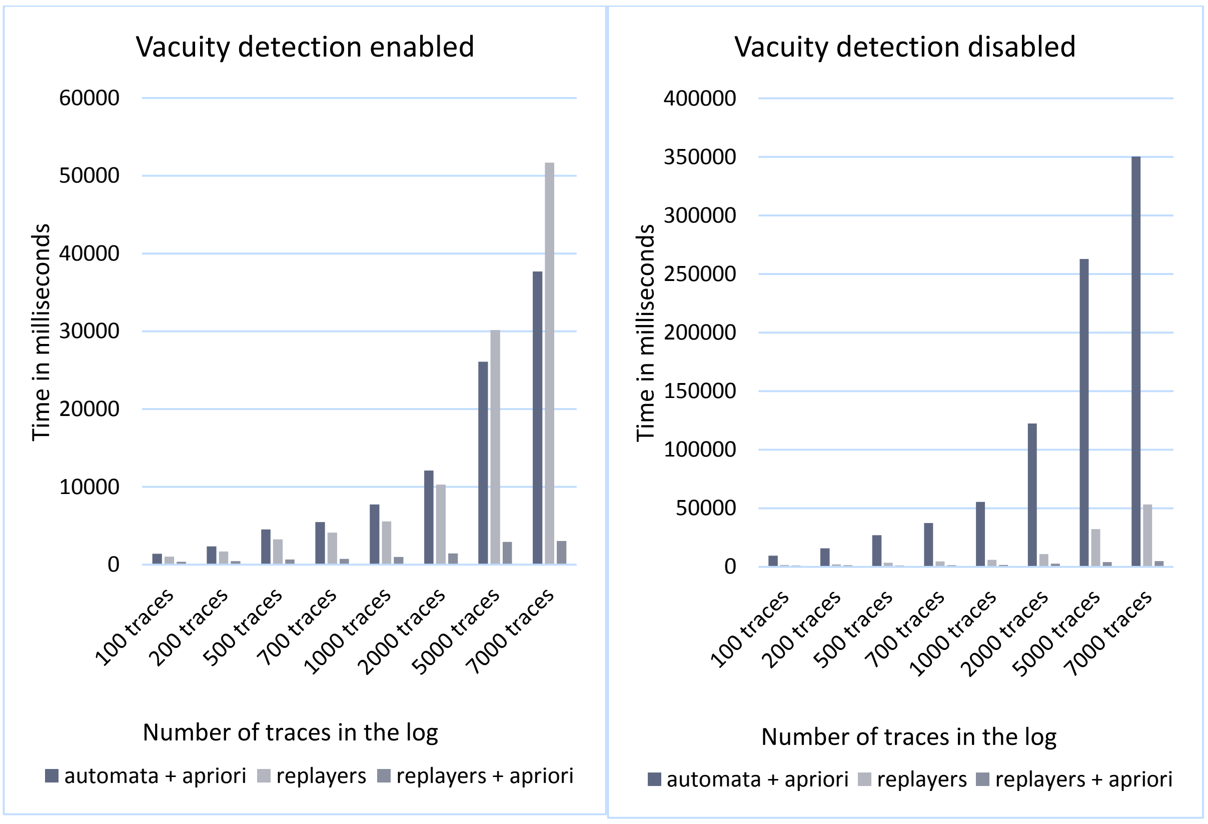


Figure 7: Log size results for minimum support 90%

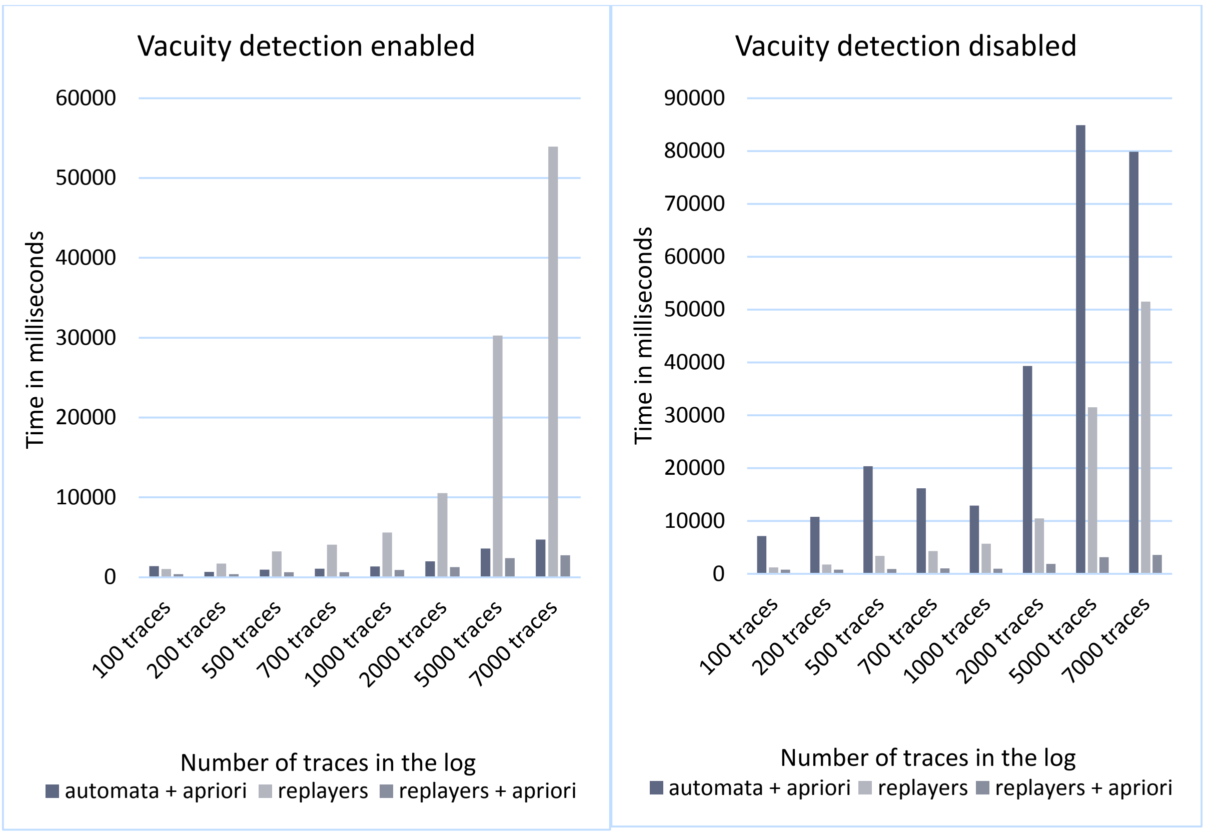


Figure 8: Log size results for minimum support 100%

Varying the trace length

The results of the following logs, which were generated by changing the trace length and fixing log size to 1000 and alphabet to 20, are presented based on the minimum support and alpha values (vacuity detection) as well.

Similarly to previous results, based on the log size, trace length benchmarks are much better for the replayers and apriori. Disabling the vacuity detection makes the automata and apriori very slow compared to the replayers and apriori.

It can be seen from the 100% minimum support graphs that the results for the automata and apriori in case of 5, 10 and 15 events per trace are quite good, outperforming the replayers only implementation in case of enabled vacuity detection. This also applies to the disabled vacuity detection but only for 5 and 10 events per trace. The replayers and apriori shows steady results that are faster than both other implementations in any case.

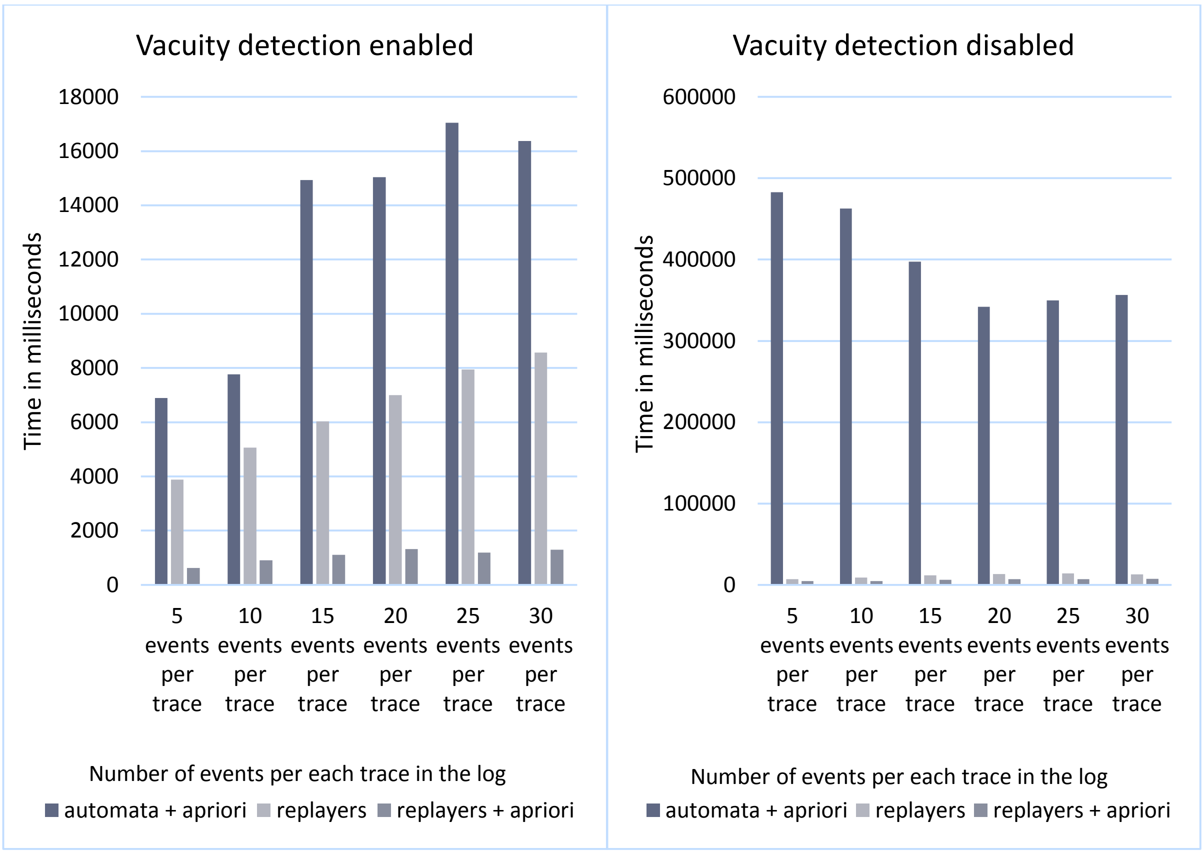


Figure 9: Trace length results for minimum support 60%

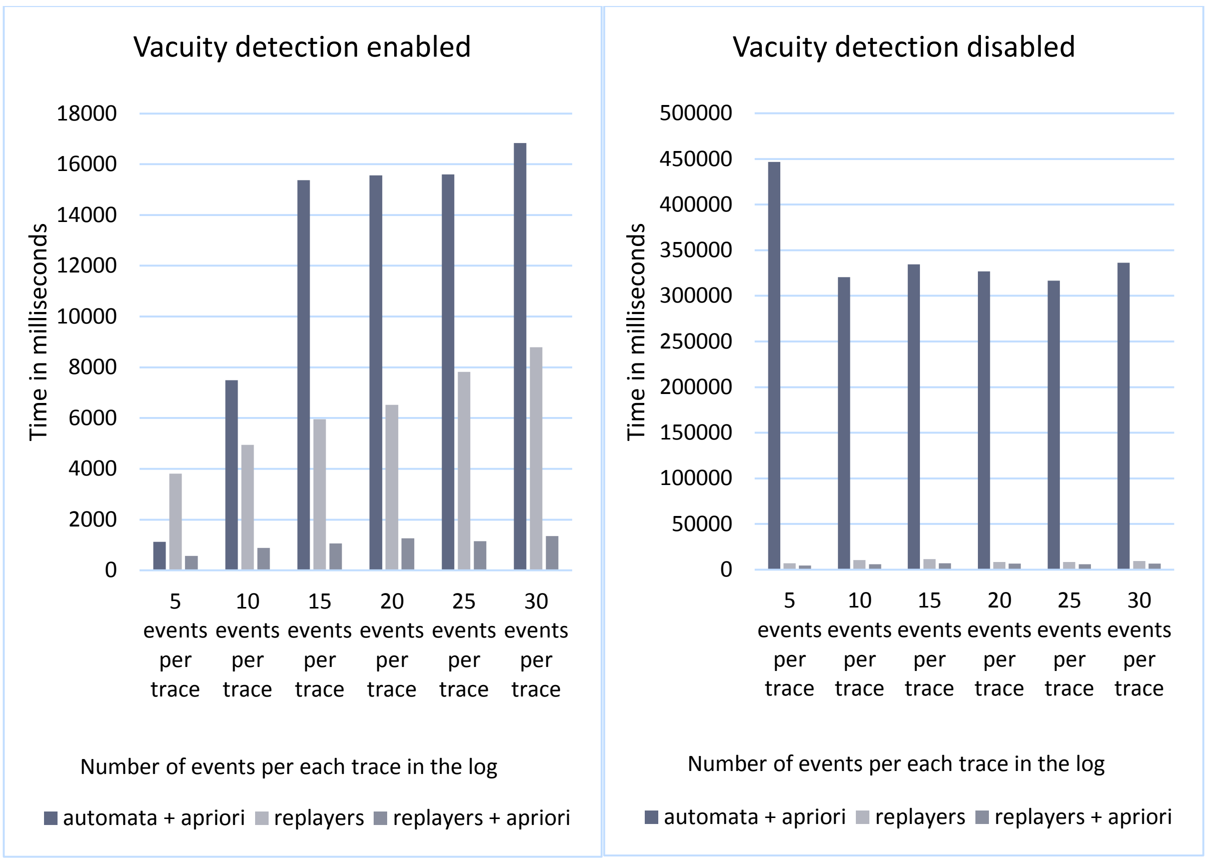


Figure 10: Trace length results for minimum support 70%

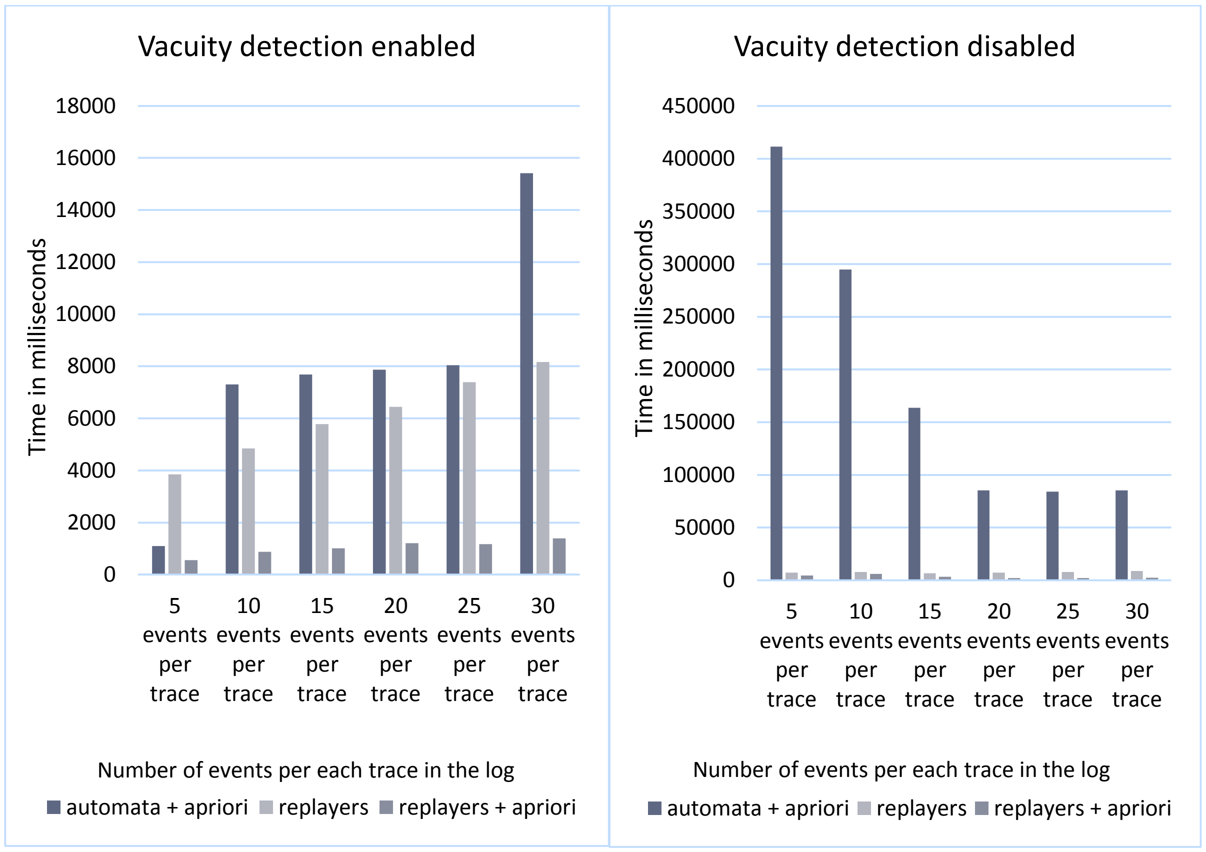


Figure 11: Trace length results for minimum support 80%

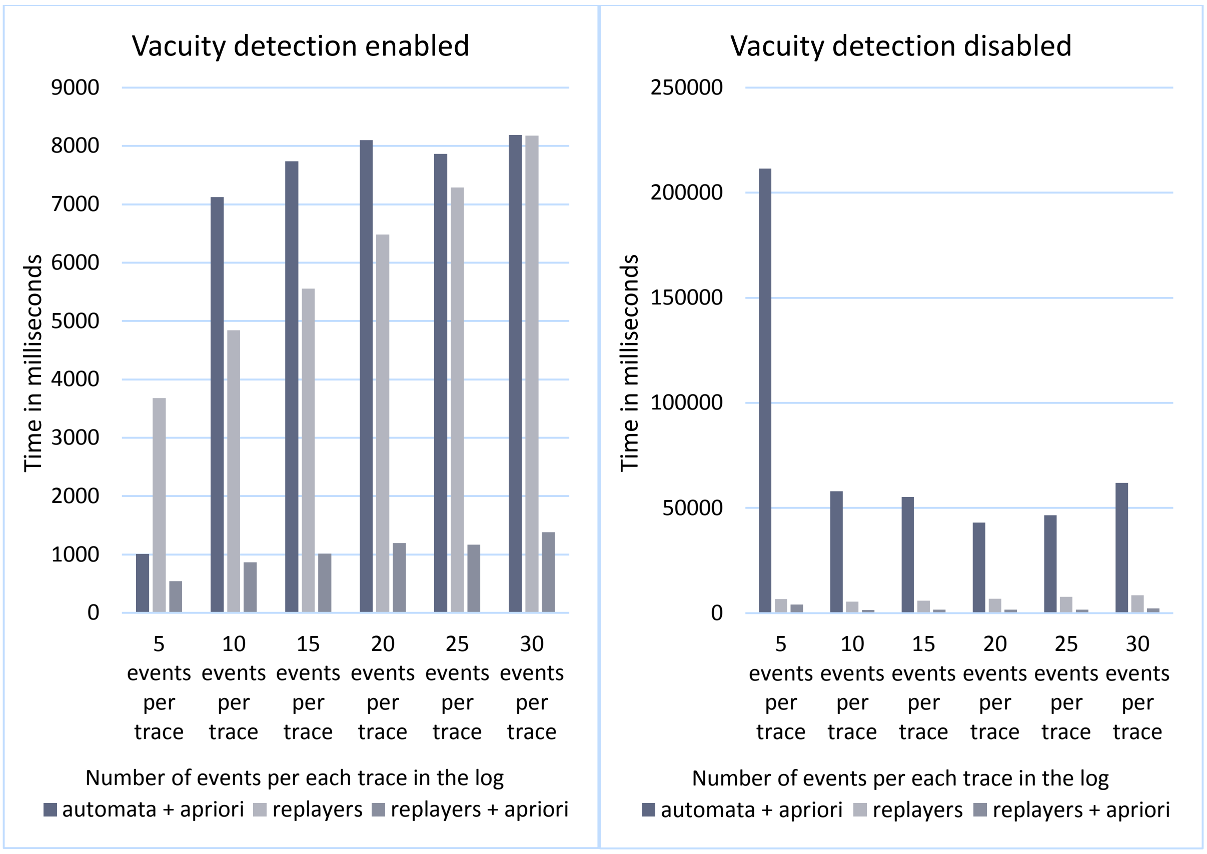


Figure 12: Trace length results for minimum support 90%

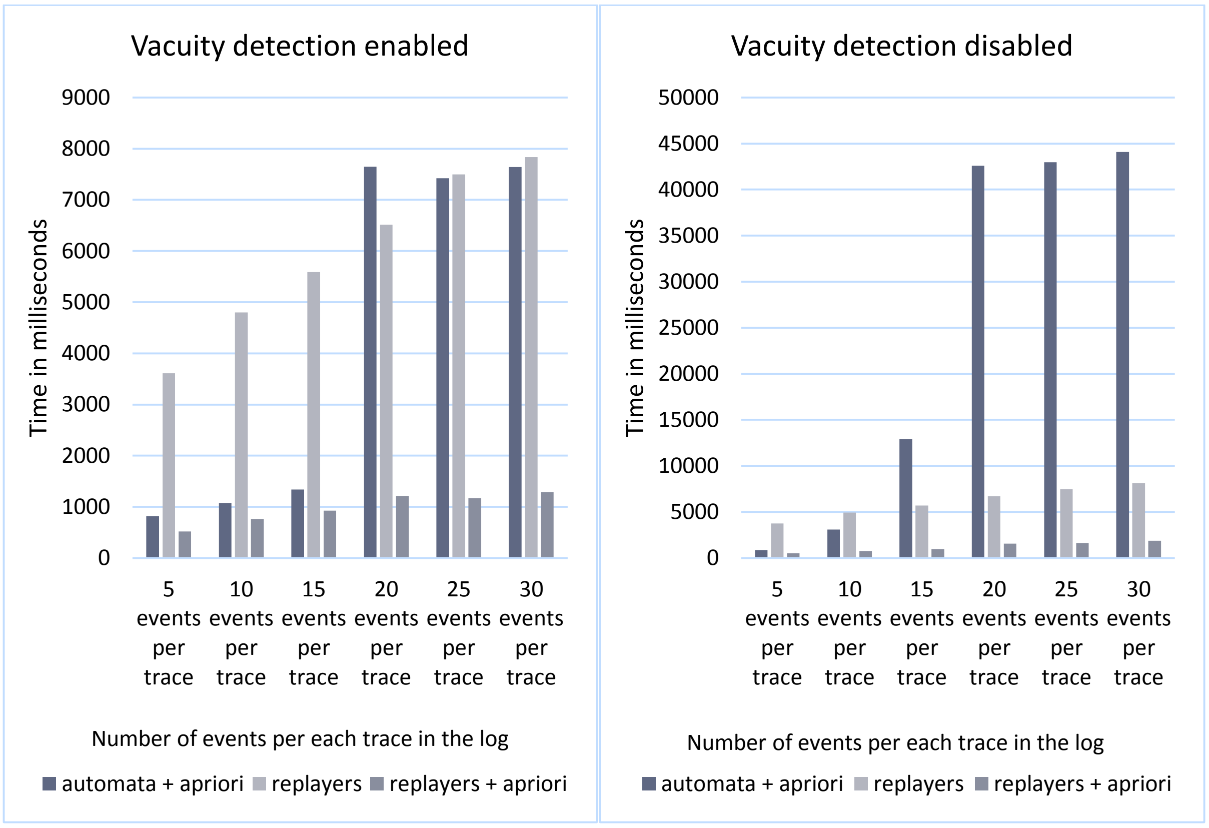


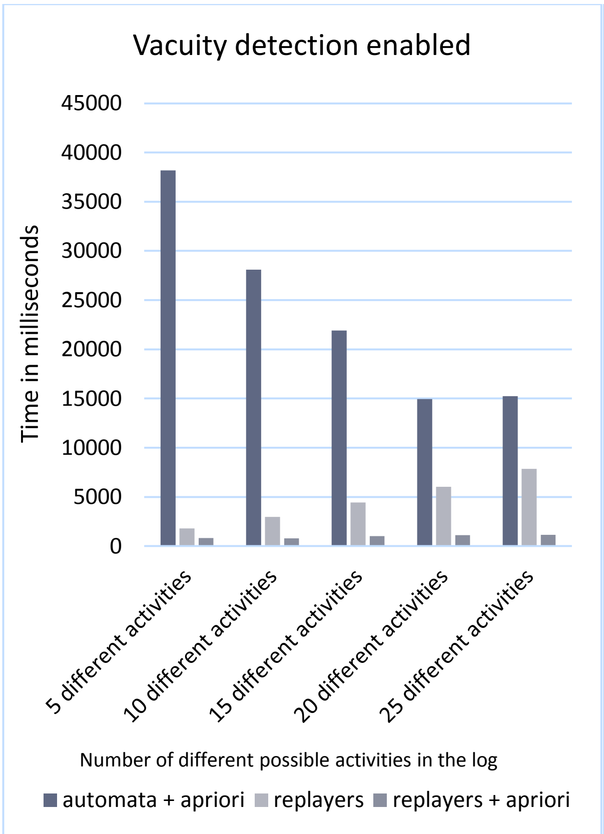
Figure 13: Trace length results for minimum support 100%

Varying the alphabet size

The third set of synthetic logs that were tested are the ones where the alphabet size varied from 5 to 25 with 5 point increments while trace length was fixed to 15 and log size to 1000. The results for this set are also presented for all the possible minimum support values and both enabled and disabled vacuity detection values.

The results show that both the replayers and apriori and the replayers only implementation are faster than the automata and apriori in both vacuity detection cases. It can also be seen from figure 14 that the automata and apriori gains performance as the alphabet size increases in case of enabled vacuity detection.

The automata and apriori for minimum support 100% performs rather well compared to the replayers only implementations in case of enabled vacuity detection. The automata and apriori is able to keep a steady 1 second execution time in nearly all cases while for the replayers only version the execution time increases linearly. For disabled vacuity detection the results are worse for the automata and apriori. Nevertheless, the replayers and apriori still outperforms the automata and apriori and the replayers only implementation in all cases as can be seen from figure 18.



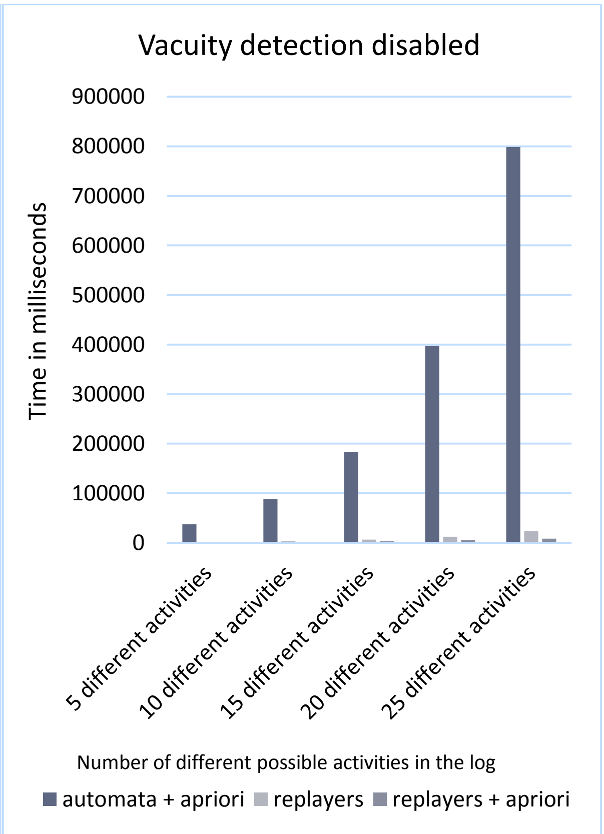


Figure 14: Alphabet size results for minimum support 60%

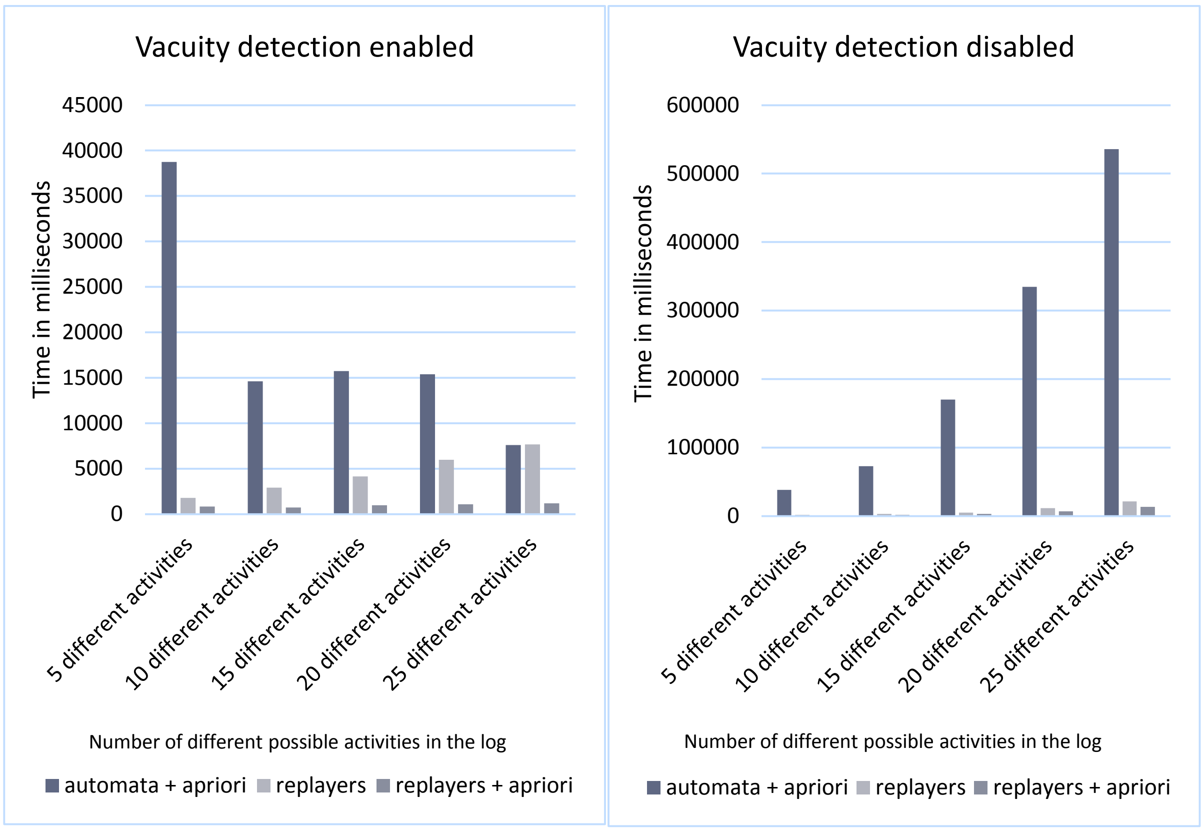


Figure 15: Alphabet size results for minimum support 70%

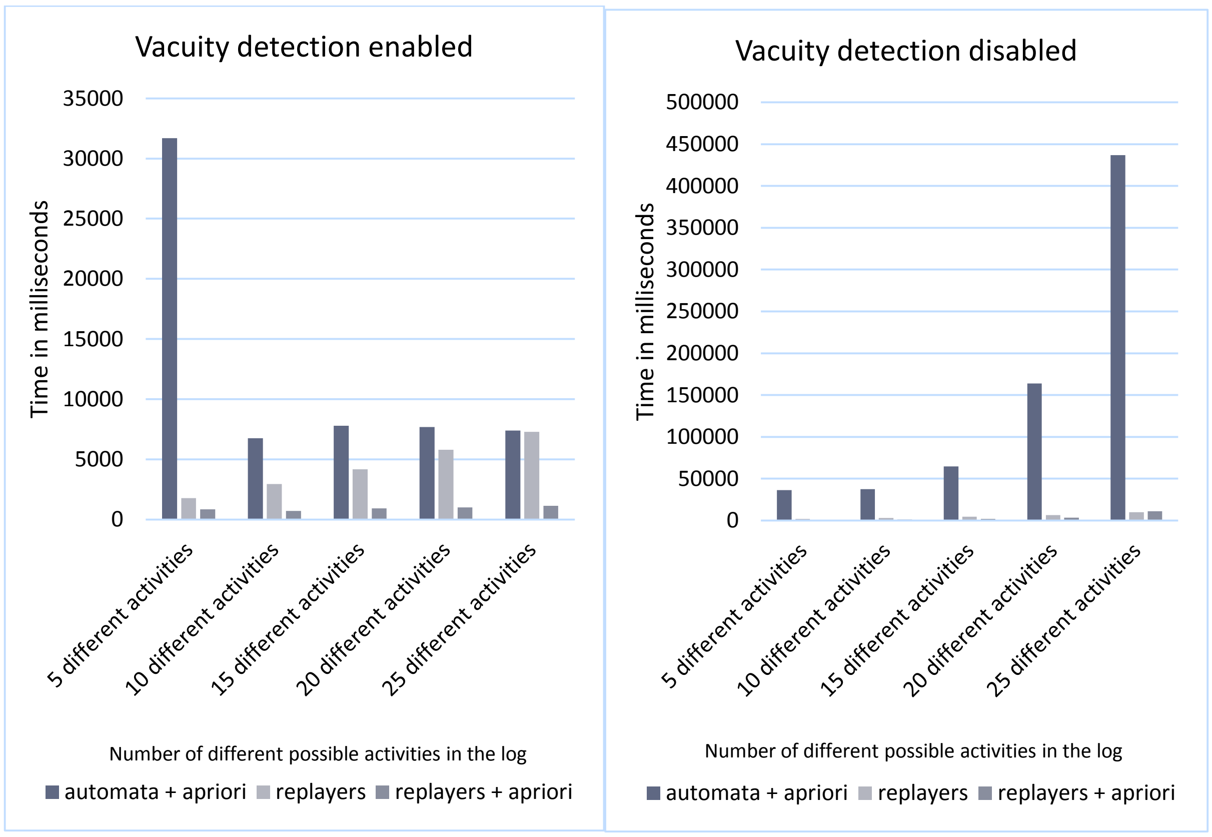


Figure 16: Alphabet size results for minimum support 80%

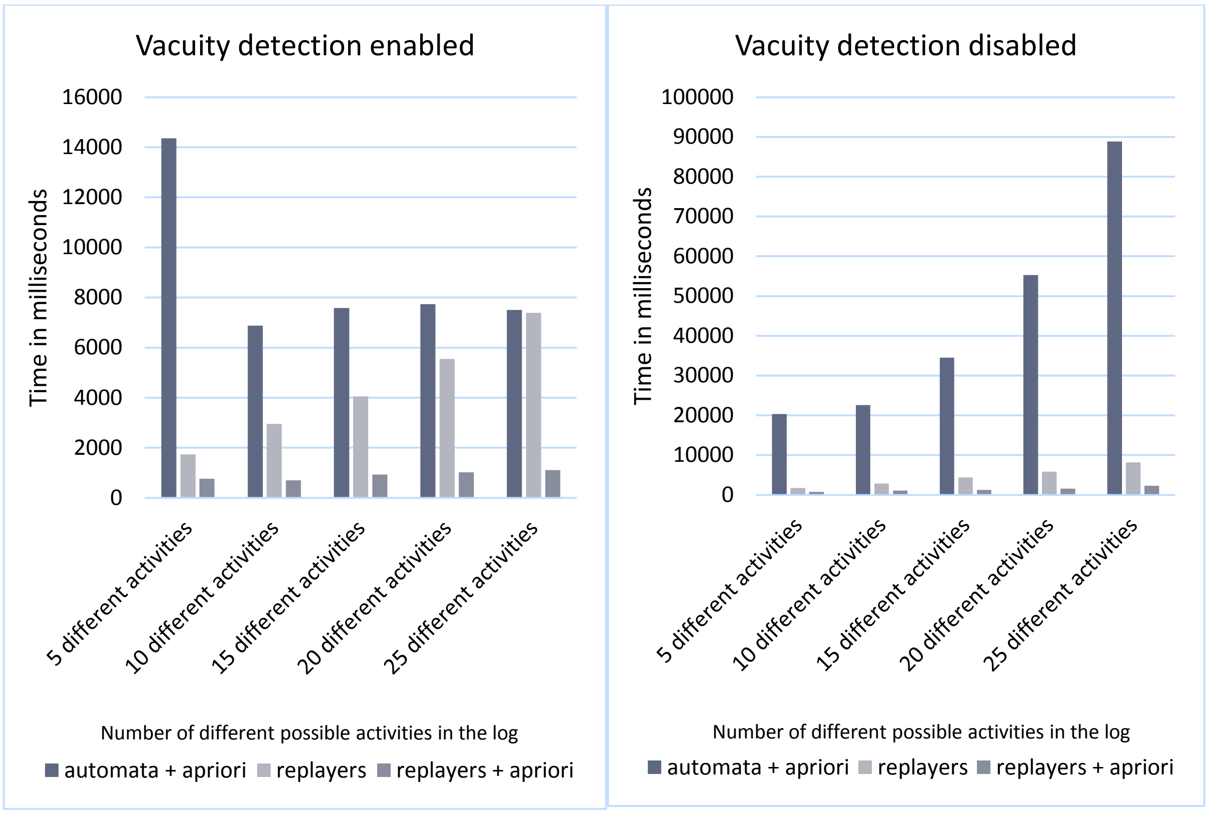


Figure 17: Alphabet size results for minimum support 90%

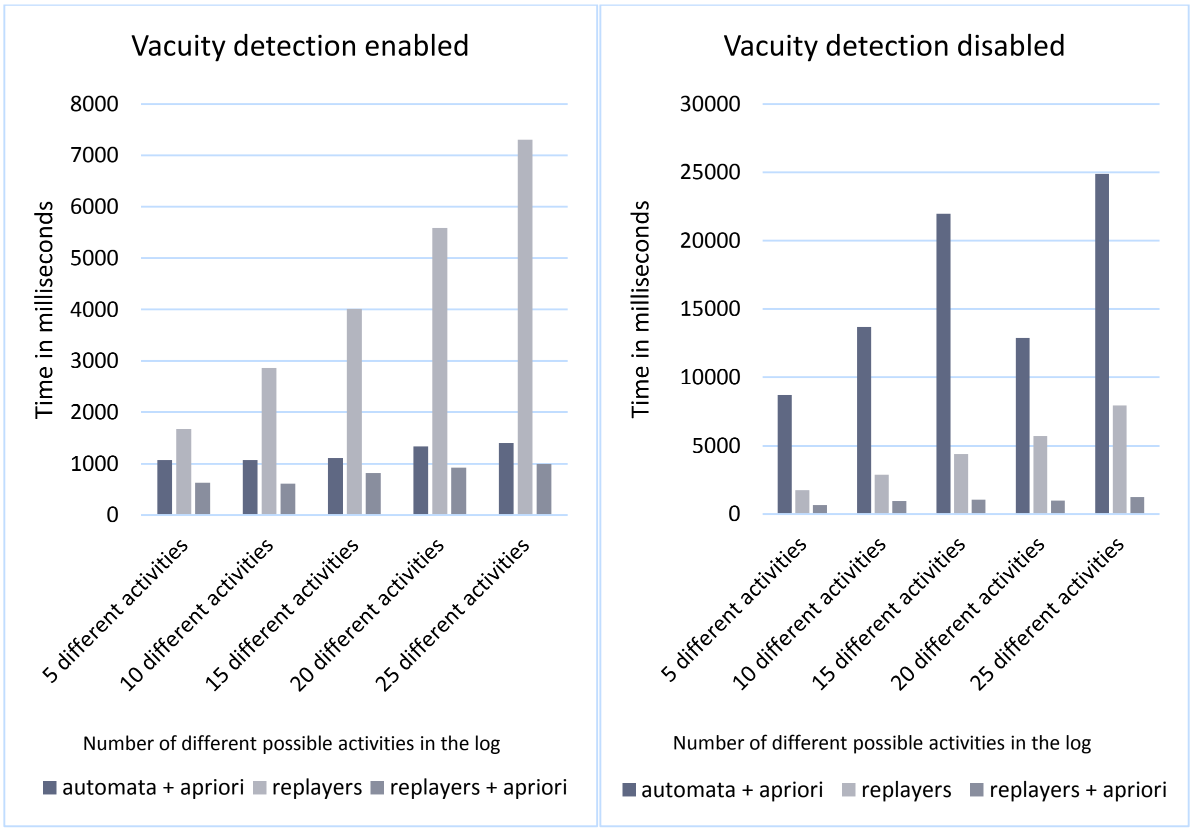


Figure 18: Alphabet size results for minimum support 100%

Template variation

Finally in the synthetic log results, the test results for single templates and with all templates together can be seen in figures 19 and 20 with vacuity detection enabled and disabled respectively. These graphs have been scaled logarithmically due to significant differences in the test result times especially for disabled vacuity detection. The execution time for each template varies but the most important observation is that the replayers and apriori is showing better results than its counterparts in most cases especially in case of all templates together. For cases where the automata and apriori was faster than the new implementation the actual difference in the execution time was a maximum of 100 milliseconds which can be considered not so relevant. These cases are Absence2 and Absence3 constraints in both vacuity detection cases. The fastest implementation for these two constraints as well as Absence turned out to be the replayers only implementation for enabled vacuity detection. For disabled vacuity detection, the replayers only implementation also showed better results than the new implementation for Alternate Precedence, Alternate Response, Chain Precedence, Chain Response, Exactly1, Exactly2 and Init constraints. The results for template variation are provided for alphabet size 20, log size 100, trace length 15 and minimum support 80%.

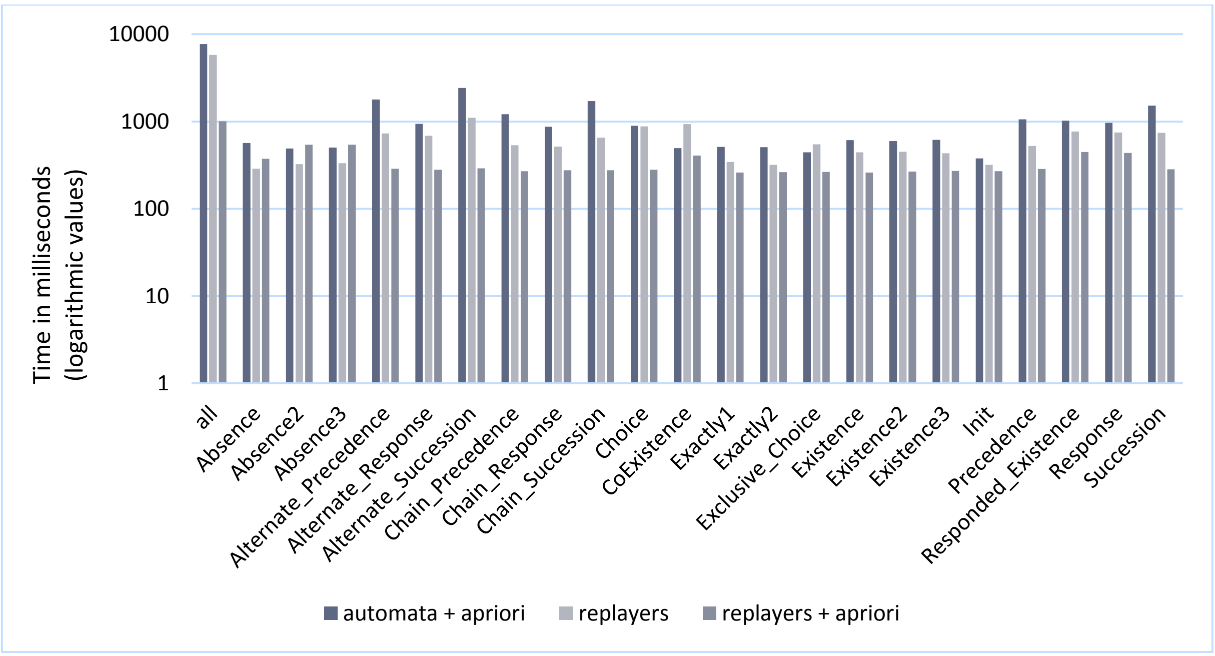


Figure 19: Template variation results for minimum support 80% and enabled vacuity detection

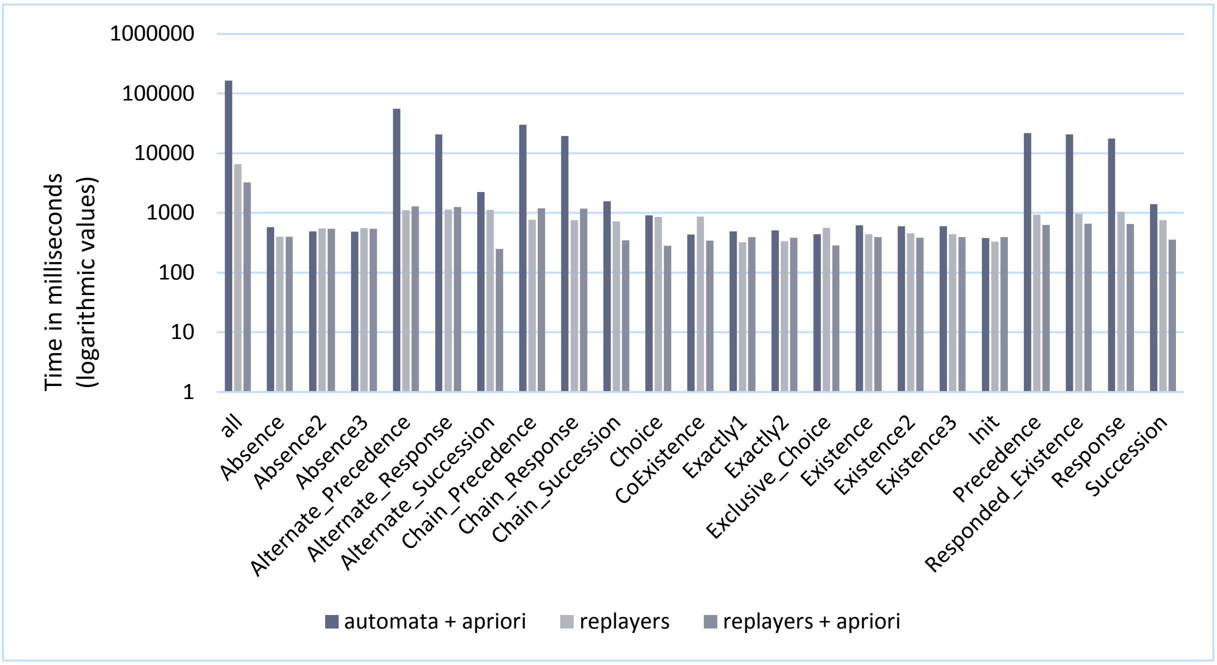


Figure 20: Template variation results for minimum support 80% and disabled vacuity detection

5.2 BPI Challenges

BPI Challenges are a bit more hefty logs that contain a random number of traces, trace lengths and alphabets. They consist of real-life event logs recorded during the execution of business processes in different context. Due to the fact that the old DeclareMiner implementation would have ran for several days for some configurations, they are not always explicitly specified.

2012 year challenge results

The 2012 year BPIC Challenge log was taken from a Dutch Financial Institute and contains 262200 events in 13087 traces. The graphs for this challenge feature results for different minimum support values and both vacuity detection cases as shown in figures 21 and 22. The automata and apriori would have taken several days to finish with the tests with disabled vacuity detection so only the replayers and apriori and replayers only implementation results are provided for this configuration.

Figure 21 shows that even for real life logs the new replayers implementation with apriori results in significant improvement over the automata and apriori with enabled vacuity detection. The average value for the replayers and apriori stays around 11 seconds compared to the average result of 2 minutes recorded for the automata and apriori implementation and the average result of 5 minutes for the replayers only implementation.

The graph for disabled vacuity detection only features the results for the replayers and apriori and the replayers only implementation. The automata and apriori would have taken several hours to complete the whole results, replayers version handled it in under 6 minutes but the replayers and apriori gave even better results, at a maximum of 1 and a half minute and a minimum of 30 seconds.

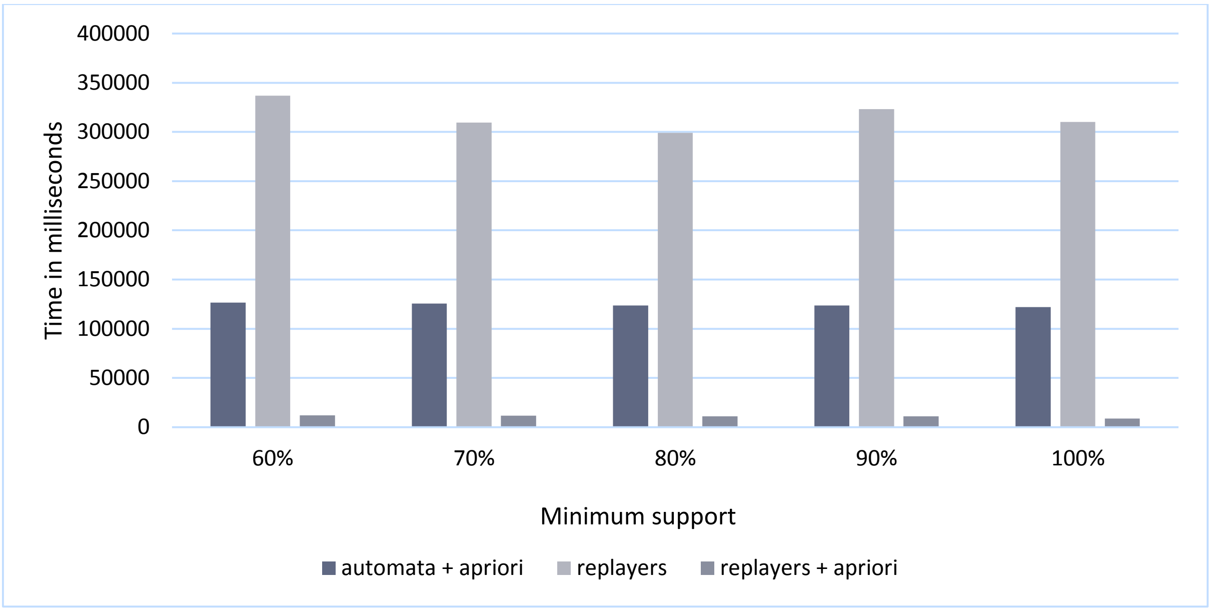


Figure 21: BPI Challenge 2012 minimum support result for enabled vacuity detection

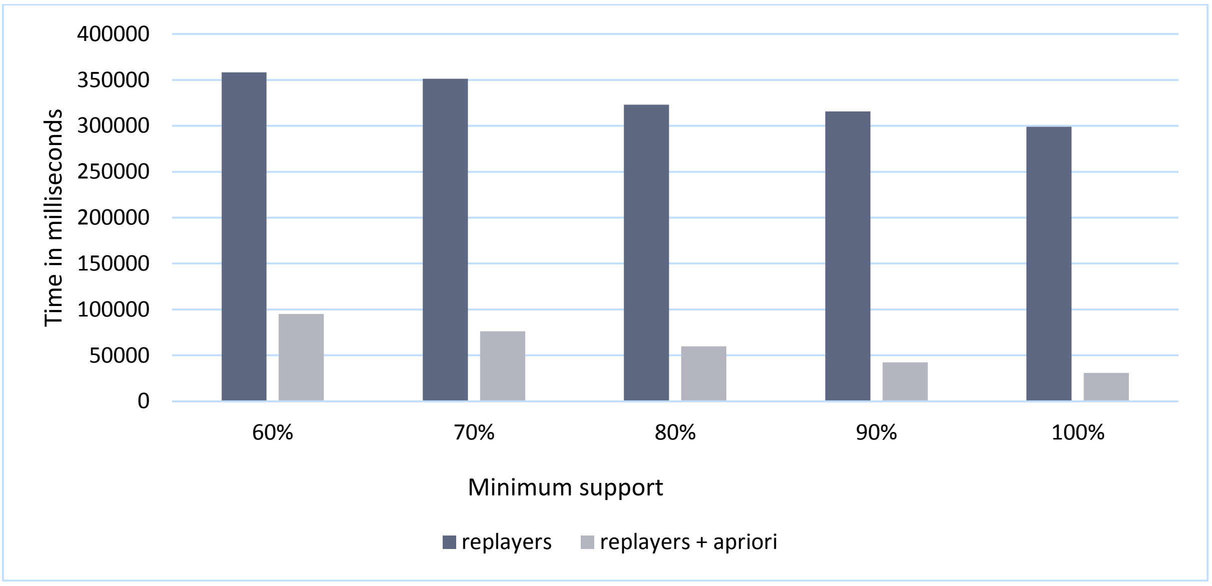


Figure 22: BPI Challenge 2012 minimum support results for disabled vacuity detection

2013 year challenge results

The 2013 year BPI Challenge log was taken from Volvo IT Belgium branch and it contains events from an incident and problem management system called VINST. The number of events in this log is 65533 and the number of traces 7554.

Figure 23 shows that the replayers and apriori has superior performance in both vacuity detection cases. Note that the automata and apriori took around two minutes to prune the candidates for 60% and 70% minimum support but only 26 seconds to do the same for minimum supports 80%, 90% and 100% for enabled vacuity detection. The average result for the replayers and apriori stays around 3 seconds for enabled vacuity detection. For disabled vacuity detection, the automata and apriori took 28 minutes to finish with minimum support 60% results and then dropped to 99 seconds for minimum support 100%. For disabled vacuity detection, the replayers and apriori took a maximum of 8 seconds to completely prune the candidate constraints.

2014 year challenge results

The 2014 year challenge log was taken from Rabobank Group ICT. The number of events in this log is 466737 and the number of traces 46616.

The automata and apriori would have taken several days to complete all the possible combinations of results for disabled vacuity detection. The replayers and apriori and the replayers only implementation took reasonable amount of time to complete the tests and thus results for these two implementations are provided for both disabled and enabled vacuity detection.

Even for enabled vacuity detection, the results for the automata and apriori are much slower compared to both replayers and apriori and the replayers only version. The 60% minimum support configuration took 2 and a half hours to prune the candidate constraints compared to the 31 seconds on the replayers and apriori.

Figure 25 shows the results for the replayers only version compared to the replayers and apriori version of the algorithm for disabled vacuity detection. While the replayers only version took 15 minutes to complete for the minimum support 60%, the new implementation did the same job in 18 seconds.

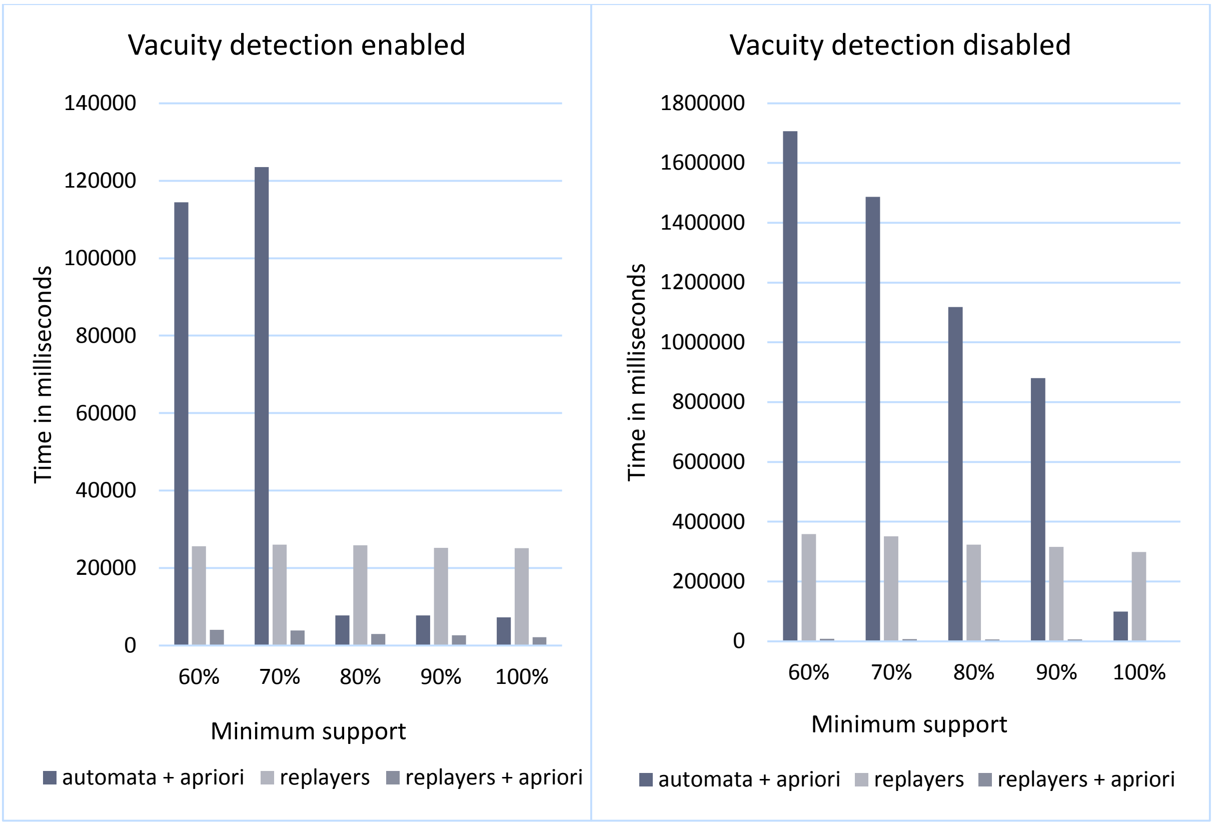


Figure 23: BPI Challenge 2013 minimum support value results

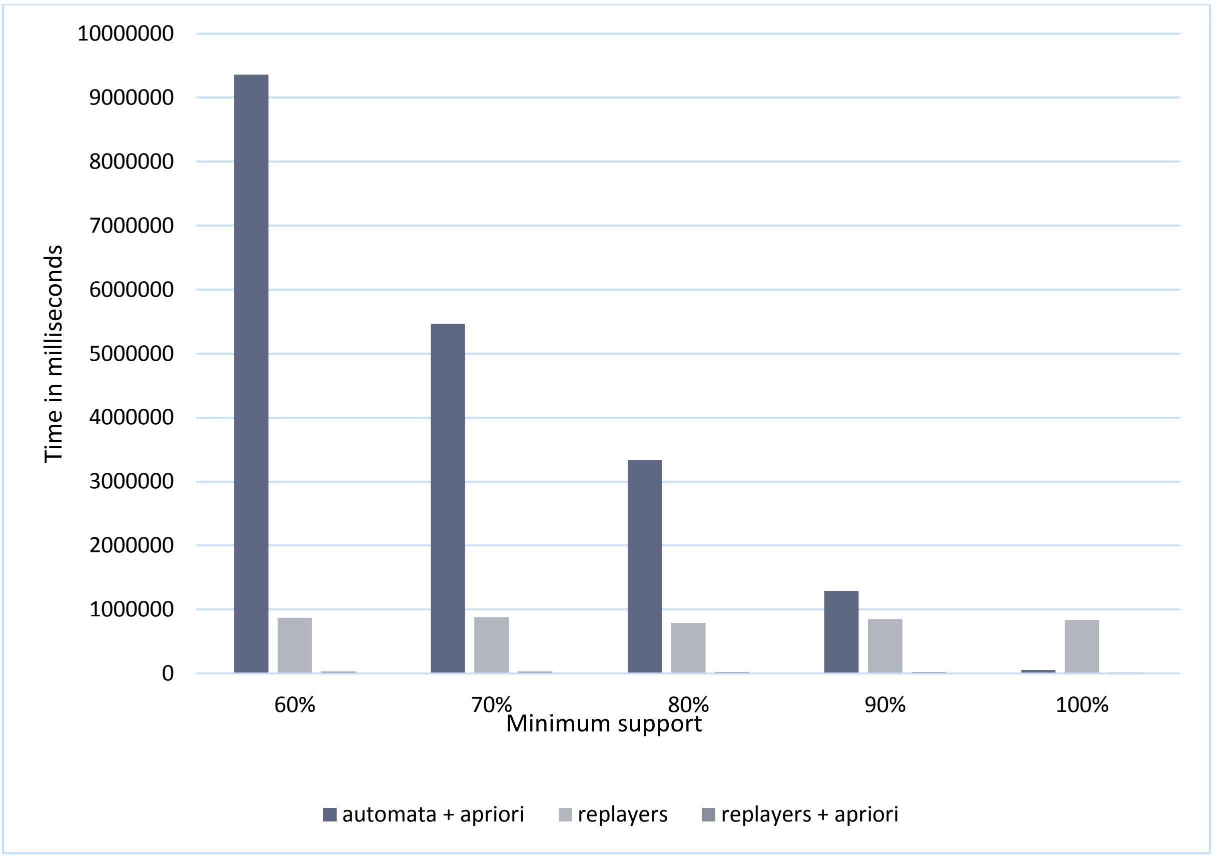


Figure 24: BPI Challenge 2014 minimum support values for enabled vacuity detection

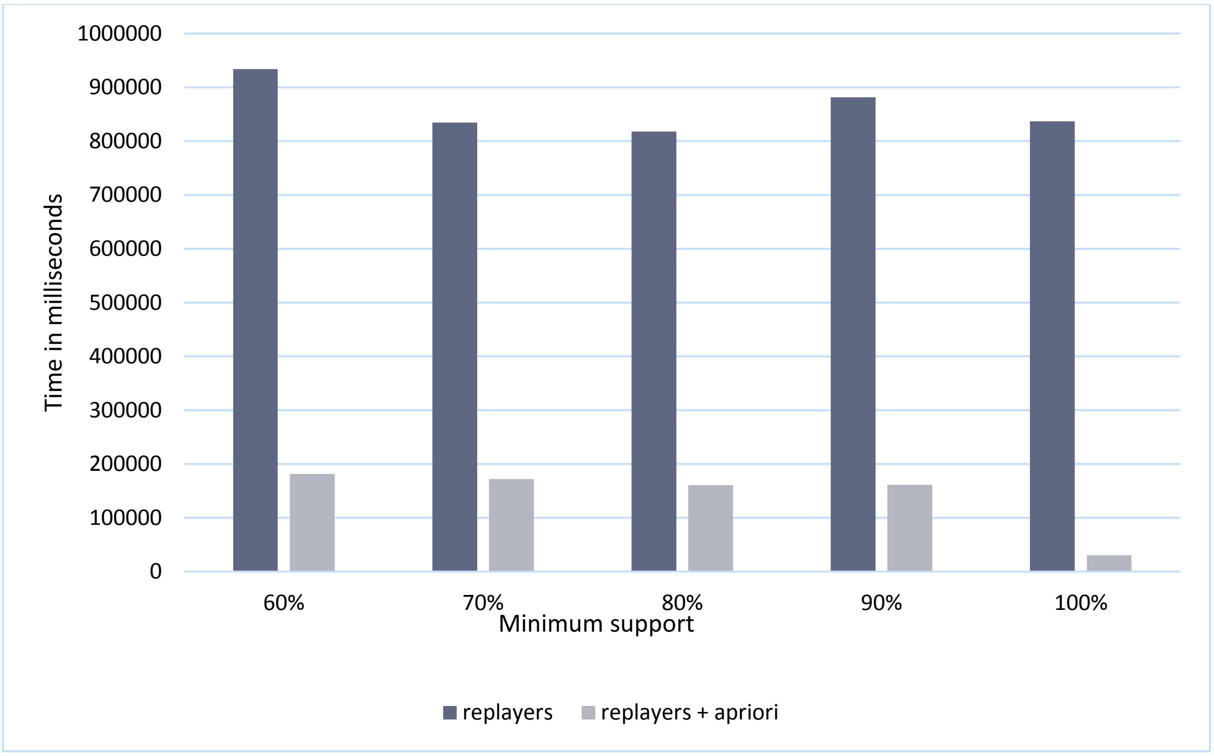


Figure 25: BPI Challenge 2014 minimum support values for disabled vacuity detection

6 Conclusion

Process mining is becoming more and more relevant for companies for analyzing their business processes. The existing solutions for declarative process mining are actively used and widely recognized but there is still room for improvement. In this thesis an existing implementation of mining declarative process models from event logs, a plugin called DeclareMiner for the ProM framework, was taken as basis and an improved version of it was developed. The new implementation uses the approach based on replayers mentioned in [1]. The algorithms based on replayers were fit into the existing algorithm implemented in DeclareMiner. The new implementation is completely separated from the ProM framework. It is a standalone application that can easily be used as a JAR file in any situation needed.

Based on the comparison results of the automata and apriori and replayers and apriori, in most of the cases tested, the new implementation outperformed the old one. There were few configurations where the old implementation was performing better but the difference was very minimal and can be considered irrelevant. Disabling vacuity detection showed the largest difference in results.
On top of improved performance, the new implementation of DeclareMiner gives the user the ability to choose between three different output types. Instead of having only one option of output, the previous XML file that was only readable by specific Declare Designer software, the new implementation is also able to generate simple text files containing all discovered constraints or a fully human-readable office report where one can see the discovered constraints in a manner that is understandable to a person not familiar with Declare.

For future work, multiple ideas are currently available. One of the ideas is to take the newly developed standalone application and fit it into a web application called RuM so it would be more widely accessible. Another idea is to extend the supported configurations such as the ability to repair existing models and adding support for additional declare perspectives such as time and data.

7 References

1. Maggi, F.M., et al. *Online process discovery to detect concept drifts in ltl-based declarative process models*. in *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*. 2013. Springer.
2. Van Der Aalst, W., *Process mining: discovery, conformance and enhancement of business processes*. 2011: Springer Science & Business Media.
3. Van Der Aalst, W., et al. *Process mining manifesto*. in *Business process management workshops*. 2012. Springer.
4. Maggi, F.M., R.J.C. Bose, and W.M. van der Aalst. *Efficient discovery of understandable declarative process models from event logs*. in *Advanced Information Systems Engineering*. 2012. Springer.
5. Pesic, M., *Constraint-based workflow management systems: shifting control to users*. 2008, Technische Universiteit Eindhoven.
6. Bellodi, E., F. Riguzzi, and E. Lamma, *Probabilistic declarative process mining*, in *Knowledge Science, Engineering and Management*. 2010, Springer. p. 292-303.
7. Bellodi, E., F. Riguzzi, and E. Lamma. *Probabilistic Logic-Based Process Mining*. in *CILC*. 2010.
8. Chesani, F., et al., *Exploiting inductive logic programming techniques for declarative process mining*, in *Transactions on Petri Nets and Other Models of Concurrency II*. 2009, Springer. p. 278-295.
9. Di Ciccio, C. and M. Mecella. *A two-step fast algorithm for the automated discovery of declarative workflows*. in *Computational Intelligence and Data Mining (CIDM), 2013 IEEE Symposium on*. 2013. IEEE.
10. van Dongen, B.F., et al., *The ProM framework: A new era in process mining tool support*, in *Applications and Theory of Petri Nets 2005*. 2005, Springer. p. 444-454.
11. Verbeek, H., et al., *Xes, xesame, and prom 6*, in *Information Systems Evolution*. 2011, Springer. p. 60-75.
12. Maggi, F.M., A.J. Mooij, and W.M. van der Aalst. *User-guided discovery of declarative process models*. in *Computational Intelligence and Data Mining (CIDM), 2011 IEEE Symposium on*. 2011. IEEE.
13. Pesic, M., H. Schonenberg, and W.M. van der Aalst. *Declare: Full support for loosely-structured processes*. in *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*. 2007. IEEE.
14. Di Ciccio, C. and M. Mecella, *MINERful, a mining algorithm for declarative process constraints in MailOfMine*. Department of Computer and System Sciences Antonio Ruberti Technical Reports, 2012. **4**(3).

8 Appendices

8.1 License